Why you should develop for CD-32, page 3.

DISK ENCLOSED

AC's TECH For The Commodore AMIGA

Volume 3 Number 4
US $14.95 Canada $19.95

TIME-EFFICIENT ANIMATIONS

- •F-BASIC 5.0
- •Quick Menus for True BASIC Programs
- •Time Efficient Animations
- •True BASIC Input Mask
- •Programming the Amiga in Assembly Language
- •Re Color
- •3-D Graphics Package Part II

Special Offer for AC Readers!

AMOS (US), AMOS Compiler, and AMOS 3D

all three for only $99.99*

Bring your Amiga to *Life!*

AMOS - The Creator is like nothing you've ever seen before on the Amiga. If you want to harness the hidden power of your Amiga, then AMOS is for you!
AMOS Basic is a sophisticated development language with more than 500 different commands to produce the results you want with the minimum of effort. This special version of AMOS has been created to perfectly meet the needs of American Amiga owners. It includes clearer and brighter graphics than ever before, and a specially adapted screen size (NTSC).

"Whether you are a budding Amiga programmer who wants to create fancy graphics without weeks of typing, or a seasoned veteran who wants to build a graphic user interface with the minimum of fuss and link with C routines, AMOS is ideal for you." *Amazing Computing, June 1992*

HERE ARE JUST SOME OF THE THINGS YOU CAN DO ▼

- ▶ Define and animate hardware and software sprites (bobs) with lightning speed.
- ▶ Display up to eight screens on your TV at once - each with its own color palette and resolution (including HAM, interlace, half-brite and dual playfield modes).
- ▶ Scroll a screen with ease. Create multi-level parallax scrolling by overlapping different screens - perfect for scrolling shoot-em-ups.
- ▶ Use the unique AMOS Animation Language to create complex animation sequences for sprites, bobs or screens which work on interrupt.
- ▶ Play Soundtracker, Sonix or GMC (Games Music Creator) tunes or IFF samples on interrupt to bring your programs vividly to life.
- ▶ Use commands like RAINBOW and COPPER MOVE to create fabulous color bars like the very best demos.
- ▶ Transfer STOS programs to your Amiga and quickly get them working like the original.
- ▶ Use AMOS on any Amiga from an A500 with a single drive to the very latest model with hard disk.

WHAT YOU GET!

AMOS (US)—AMOS BASIC, sprite editor, Magic Forest and Amosteroids arcade games, Castle AMOS graphical adventure, Number Leap educational game, 400-page manual with more than 80 example programs on disk, sample tunes, sprite files, and registration card.
AMOS Compiler—AMOS Compiler, AMOS language updater, AMOS Assembler, eight demonstration programs which show off the power of the compiler, and a comprehensive, easy-to-use manual to develop lightning fast software.
AMOS 3D—Object Modeler, 30 new AMOS commands, and more. AMOS 3D allows you to create 3D animations as fast as 16 to 25 frames per second. You can display up to 20 objects at once, mix 3D with other AMOS features such as sprites, bobs, plus backgrounds, and more.

Limited Time Offer for AC readers only!

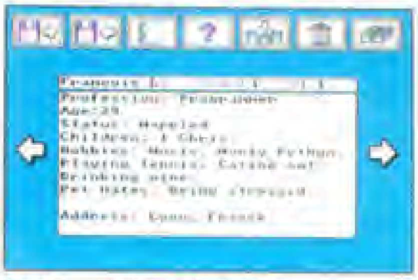
Get all three AMOS packages at one great price. Order today by sending your name, address (physical address please—all orders will be shipped by UPS), and $99.99 (*plus $10.00 for Shipping and handling) to: AMOS Special, PiM Publications, Inc., P.O. Box 2140, Fall River, MA 02722-2140 or use your VISA, MasterCard, or Discover and fax 1-508-675-6002 or call toll free in the US or Canada:

1-800-345-3360

europress SOFTWARE

Please allow 4 to 6 weeks for delivery.
AMOS written by François Lionet
©1992 Mandarin/Jawx
Country of Origin: UK

Use the sophisticated editor to design your creations


Francois L
Profession: Programmer
Age: 29
Status: Happied
Children: 1 Chris
Hobbies: Music, Monty Python,
Playing Tennis, Eating out
Drinking wine
Pet hates: Being stressed
Address: Lyon, France

Create serious software like Dataflex

Produce educational programs with ease

Play Magic Forest and see just what AMOS can do!

Design sprites using the powerful Sprite Editor

Create breathtaking graphical effects as never before

Contents

Volume 3, Number 4 AC's TECH AMIGA

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:

AC's TECH Submissions
PiM Publications, Inc.
P.O. Box 2140
Fall River, MA 02722-2140

AC's TECH AMIGA

ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Administrative Asst.:	Donna Viveiros
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Traffic Manager:	Robert Gamble
Marketing Manager:	Ernest P. Viveiros Sr.

EDITORIAL

Managing Editor:	Don Hicks
Editor:	Jeffrey Gamble
Hardware Editor:	Ernest P. Viveiros Sr.
Senior Copy Editor:	Paul Larrivee
Copy Editor:	Elizabeth Harris
Video Consultant:	Frank McMahon
Illustrator:	Brian Fox

ADVERTISING SALES

Advertising Manager: Wayne Arruda

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

AC's TECH For The Commodore Amiga™ (ISSN 1053-7929) is published quarterly by PiM Publications, Inc., One Currant Road, P.O. Box 2140, Fall River, MA 02722-2140.

Subscriptions in the U.S.: 4 issues for $43.95; in Canada & Mexico surface: $52.95; foreign surface for $56.95.

Application to mail at Second-Class postage rates pending at Fall River, MA 02722.

POSTMASTER: Send address changes to PiM Publications Inc., P.O. Box 2140, Fall River, MA 02722-2140. Printed in the U.S.A. Copyright© 1991 by PiM Publications, Inc. All rights reserved.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

AMIGA™ is a registered trademark of Commodore-Amiga, Inc.

An Interview with Commodore's Chris Ludwig

Developing for the Amiga CD^{32}

Immediately after the release in Europe of the Amiga CD32 and in preparation for the North American release, AC's TECH was fortunate enough to contact Chris Ludwig, Commodore's Multimedia Standards Engineer and Wayne Lutz, of Technical Support, to discuss some of the features and possibilities of Commodore's new game platform. Below is a series of questions and answers concerning CD32 features and development potential.

AC: *What makes the CD32 different from the original CDTV?*
Ludwig: CD32 has a lot more features than the CDTV had. It is more up-to-date with the Amiga line, with a full AA chip set and a 14MHz '020. The top-loading CD-ROM drive is dual speed so it will play at 300K per second as well as 150K per second and does not require a caddy. It has fewer ports than the CDTV, which were removed as a cost reduction measure. It is squarely targeted at the same entertainment audience as a SEGA CD in that it needs to be low cost and does not require computer features. It has game features.

AC: *How is CD32 different from other platforms?*
Ludwig: Versus other platforms, the benefits are obvious. There are over 4 million Amigas in the world and each is potentially a CD32 development system. Amiga developers have been creating products since 1985. There is plenty of support software available to help developers create software.

As far as hardware, the CD32 is technically superior to anything that is currently on the market. It is the only 32-bit CD games console as well as the only game console available with a 300K per second CD drive. The graphic quality provided by the AA chip set offers better graphics than anything else with higher resolution displays and many more colors than the SEGA or Nintendo systems.

AC: *What about the competition from 3DO?*
Ludwig: CD32 has a higher graphic resolution than 3DO. We have a part called the AKIKO which is a fairly large gate array that includes a number of the glue functions of the 1200.

The AKIKO also has a part called the chunky-to-planar converter which allows developers to work with 3-D data a lot faster than they would on a traditional Amiga system, even faster than on an Amiga 1200, bringing us closer to the kinds of things 3DO has promised and ours is a lot cheaper.

AC: *The price in the U.S. will be?*
Ludwig: The only price I am sure of is the £299 announced in the UK. I am sure the price will be set for North America soon.

AC: *What is the capacity of the CD ROM?*
Ludwig: CD32 uses a standard ISO 9660 file system. That translates to about 580MB minus 150K per second audio or MPEG data.

AC: *What features in CD32 help in cross-platform work?*
Ludwig: If you are doing cross-platform work, one of the nicest things is that CD32 has native modes which are very similar to the native modes of other machines—especially PC machines. A lot of developers do their work for PCs first and then scale down to a console. With CD32 they would not need to scale their work down. If they were starting with 640 x 400 or 640 x 480 256-color graphics, they could use those graphics directly. It would be a matter of converting them to Amiga but they would not lose any resolution. In fact, the chunky-to-planar hardware can convert the pictures for you.

AC: *There is no CD32 expansion yet announced from Commodore for existing Amigas?*
Ludwig: That is true. We have started to add software support for the CD32 in the Workbench 3.1 which is not yet available. Almost all of the extra bits in the CD32 ROM such as the low level library and the CD.device will be available as disk-loadable libraries when 3.1 is released. In fact, using the CD device and CD file system, most CD32 titles should run with no problem on any AA chip set Amiga with a SCSI-2 CD-ROM drive.

AC: *What do you like best about the CD32?*
Ludwig: To me the expandable nature of CD32 is cool. It has a huge expansion bus that has nearly every signal in the machine except the keyboard and joysticks. That opens us up to a lot of markets. It can get to be a faster machine in a hurry. I can see developers providing extra memory and accelerators for CD32. Extra memory would act as an accelerator because the extra memory would be fast RAM and that would increase code execution speed. There is a ton of things you can add on to a port like that. It could be a cable controller box or anything else that you want to do with it.

MPEG is one expansion already planned for the port.
Ludwig: Wayne what do you like about the CD32?
Lutz: I like the double speed drive in connection with the AA chip set. Because you can pull data off the drive very fast and then display it on the screen with 16 million colors, it looks really sharp. It opens a lot of opportunities for games, entertainment, and educational titles.
Ludwig: The controller is really nice too. The extra buttons allow a lot more flexibility when you are designing games. Most games today require a lot more control than can be afforded by four directions and a fire button.

AC: *This can be much more than a game machine?*
Ludwig: There are certain areas where it will be ideal. Basically anywhere where you need to install a lot of delivery platforms. Because of the machine's low cost and its ability to hold a lot of data, with an MPEG module, or even without it, it would be ideal as a replacement for laser disk video systems that are being installed in kiosks nation-wide these days. CD32 is ideal for these types of installations. The high graphic quality AA chip set lets you supply photographic quality that will present a product very well.

It makes sense to consider this box for a cable television controller. For anything concerning video in the home, it is perfect: low cost, lots more processing power, lots more memory, and a good delivery platform.

AC: *Most of the development is happening in Europe?*
Ludwig: For the most part. It is surprising for people to learn just how many games are developed in Europe, especially in the UK.

AC: *Any additional comments?*
Ludwig: There is one thing. I have John Campbell's (Commodore's CATS director) five steps to Amiga success on the CD32 from a letter going to developers.

Step 1. Keep your developer status current. If you are not a developer than you should become one.

Step 2. There is a publication available to registered developers called the *Amiga CD32 Developer Notes* which would be useful.

Step 3. There is a licensing agreement required for anyone who wants to create Amiga CD32 disks and distribute them. This is what is needed to get development tools for the CD32.

Step 4. As far as a developer system, it is important that you get an AGA-based Amiga. At some point you will want to invest in a CD-ROM writer. There is nothing like saying it is done than handing your grandmother a disk to play and experiment with.

Step 5. Join BIX. The developer conferences for registered developers are important and it is the place to get electronic tools and electronic versions of documentations. It is a great distribution system for us.

AC: *CBM's licensing agreement is a lot easier to deal with than other vendors.*
Ludwig: That is true. We do not require developers to come to us to have their disks pressed, while Nintendo and SEGA require developers to have their product duplicated and produced by them. This adds considerably to the production cost. ☑

F-Basic 5.0

Some New Windows on the Amiga Scene

by Jeffrey Stein

F-Basic 5.0 is the latest and most mature version in the evolution of a powerful programming language for the Amiga. Over the last five years, each new version of F-Basic from Delphi Noetic Systems, Inc has removed restrictions and enhanced the scope of this system.

Version 1.0, released in 1987, provided neither editor nor linker and, among other limitations, ran only from the CLI. Version 2.0, released by DNS in the spring of 1989, added the ability to run under the WorkBench, along with providing its own linker to allow standalone executable object files. Additionally, high level reading and writing of IFF picture files, random access files, high level animation support, and double precision floating point were included. Next, an integrated editor environment, direct 68020/68881 support, IFF sound file player, user-defined operations on record structures, and a built-in matrix, and complex number package appeared in Version 3.0 (circa Fall, 1990). With the advent of F-Basic 4.0 early in 1992, DNS developed high level ARexx support, gadget and advanced mouse features, separately compiled modules, 68030/68882 support, and an improved editor and WorkBench icon arguments in the language.

This article is based on a pre-release version of F-Basic 5.0, obtained from Delphi Noetic. According to their spokesperson, the 5.0 release is scheduled for spring 1993 and probably will be shipping by the time this review is published. For those unfamiliar with the F-Basic system, a brief overview of the language is first presented. This is followed by a discussion of the new features added in 5.0, most importantly the complete support provided for the various screen modes available on the ECS and AGA chip sets.

An Overview of the F-Basic Language

F-Basic is a synthesis that retains much of the syntactical simplicity of traditional BASIC while providing advanced features like record structures, pointers, recursion, access to the Amiga ROM Kernel libraries, etc. found in more modern languages like C or Pascal. At the core of F-Basic, the computation of arithmetic expressions, control structures such as FOR loops, IF/THEN statements, GOTOs, and simple input and output would be immediately recognizable by any BASIC programmer.

To emphasize the similarity of the two languages at this level, Listing 1 is a simple F-Basic program to compute a mortgage amortization schedule. The example also illustrates most of the essential differences between F-Basic and AmigaBASIC. F-Basic programs begin with a PROGRAM statement and, like C or Pascal, require that each variable be explicitly assigned a data type at the beginning of the program. This is an expected consequence of F-Basic's provided ability for the programmer to define his or her own extended record data types. Such variable typing was unnecessary in standard BASIC, because most supported only three data types: integer, real, and text.

The other major difference, and perhaps the most controversial, lies in F-Basic's method of string handling. High level languages typically represent strings in one of three ways:

Dynamically, as in standard BASIC;
Statically with a designated termination character as in C;
or Statically without a designated termination character as in F-Basic or Fortran.

Each of these strategies has its strengths and weaknesses. Dynamic strings are certainly the simplest from the programmer's point of view. One simply declares a string variable without specifying its maximum length. Each time a text string is assigned to a string variable, the previous memory used to hold the old value of the string is deleted and enough new memory is allocated to accommodate the present length of the text string. The purpose of the string variable is to keep track of the memory, just allocated, for the text string. There is a price, however, for relieving the programmer of the task of worrying about the length of the string variable. The overhead associated with deallocating and allocating memory during each string computation will slow the whole process by as much as a factor of 10, even in compiled BASIC systems. In programs that do extensive text manipulation, this is not a trivial consideration.

The alternative to dynamic strings is to associate a fixed buffer area of specified length with each string variable. This strategy is called static string allocation because the memory location used to hold the string is fixed throughout the execution of the program, thus eliminating the need for memory allocation and deallocation. The principal drawback from a programmer's standpoint is that one must estimate in advance the maximum buffer size associated with each string variable. There are two variants in this approach. The language may or may not use a special character, called the termination character, to indicate the present end of the string within the buffer. For instance, C places a zero byte after the last valid character in the string. This makes the task of assigning a short string to a longer string (as in LongString=ShortString) easy. The contents of the shorter string is copied into the buffer for the longer string and a zero byte is placed after the short string to indicate its present size. Although the use of a termination character makes such mismatched assignments easier, it too involves restrictions. One cannot store arbitrary data in a string variable, as the data may itself contain a termination character; for example, editing non-ASCII files or transmitting and receiving data over the serial port. In addition, most substring operations with a termination character are slower and more complex than they would be otherwise.

Alternatively, F-Basic and Fortran do not use a termination character. As noted, this allows strings to contain arbitrary data and makes substring operations convenient and fast. It does, however, require when assigning a short string to a longer string that the longer string be first filled with blanks if the portion of the longer string buffer beyond the end of the shorter string is to be discarded. The conclusion is that static strings, with or without a termination character, require more effort on the part of the programmer to keep track of string lengths, but pay large dividends in increased speed of execution. This is illustrated by the speed of F-Basic's single pass compiler, which is itself written in F-Basic and obviously is very string-function intensive. Previous published articles have shown F-Basic's speed of compilation to be pace-setting, even when compared to other commercial development systems.

To give the flavor of string manipulation in F-Basic, Listing 2 shows a program that inputs a text string and prints out all possible "words" obtained by rearranging the letters of the text string. This mirrors the popular "Jumble" game that appears in many newspapers across the country.

Beyond its core, F-Basic provides a number of powerful extensions to standard BASIC. These include record structures and pointers, local and global variables, and simplified high level creation of screens, windows, menus, gadgets, and requesters. Interactions with the mouse, high level access to the serial port and animation, easy access to the Amiga's ROM Kernel library functions, the ability to add programmer libraries to the standard system libraries, built-in operations with vectors, matrices, and complex numbers, and an AREXX interface are some of the more important extensions present in F-Basic.

Although simple F-Basic programs look very much like their counterparts in AmigaBASIC, F-Basic's ability to provide record structures and pointers allows it to handle more sophisticated programming tasks in a natural way. This is best illustrated by F-Basic's interaction with the AmigaDOS operating system. Internally, the Amiga associates complex record structures to manage the details for most of its major distinctive features. Records and pointers allow F-Basic programs to communicate smoothly with Amiga ROM routines without forcing arrays and integer variables to do the job. Any BASIC programmer who has been forced to PEEK or POKE memory to simulate a record structure will appreciate this addition.

It has been recognized for a long time that the best approach to managing the design of a complex program is to divide it into a number of small modules, each of which performs a simple task. In such a case, some variables are used only within one module, while other variables represent data that must be universally accessed by all modules. Data of the first type in F-Basic are assigned a data type within the module itself and are referred to as local variables. Data variables of the second type are referred to as global variables and are declared within the global variable list of the main program. This distinction is common to F-Basic, C, and Pascal.

The sample program in Listing 2 uses global variables to facilitate communication between the main module and the Jumble subroutine.

The approach taken to the implementation of Amiga-specific features such as screens, windows, gadgets, menus, sound, etc., in languages other than F-Basic falls into one of three basic categories. Some, such as *TrueBasic*, avoid Amiga-specific interactions in the name of portability. In this case, one may wonder whether an Amiga without its special features is really what a programmer is after.

Others, such as AmigaBASIC, provide simplified high level construction of these features. The underlying nature of the graphical user interface, from the viewpoint of the operating system, is very much more complex than the details that can be easily specified in any AmigaBASIC statement. Thus, this group typically makes certain default choices in defining their underlying structures and renders customization to fit special needs somewhat difficult.

A third class of systems, such as C, takes a "no holds barred" approach and requires that the programmer include all relevant data structures that would be used if one were implementing these features in assembly language. This has the advantage of providing complete control over every one of the Amiga's details.

F-Basic attempts a synthesis of the latter two methods. On the one hand, windows, screens, gadgets, menus, and the like are typically created with a single high level statement that provides more options than the corresponding AmigaBASIC statement. In addition, F-Basic returns pointers to the underlying structures for those adventurous enough to deal with them directly. As F-Basic allows the definition of such record structures, the alteration of characteristics that were defaulted by the high level statement is not as tedious as C's requirement of defining the entire structure, starting from scratch.

One of the more pleasant features within F-Basic is provided by a profusion of event structures. An event structure is a block of high level code that is executed only when the specified event occurs. Examples of events include mouse single, double, and up/down clicks, receipt of an AREXX message, keyboard clicks, collision between animated objects, menu selects, window closing or re-sizing, and serial port communication, among others.

These structures act like a high level interrupt. When the processing associated with the event is finished, control returns back to that point in the program where execution was interrupted. This feature eliminates the need for constant checking to see if some action has been taken by the user. Many characteristics associated with each occurrence, such as the mouse coordinates in the case of a mouse event, or the menu and item number in the case of a menu selection are provided by pre-defined system variables. A SLEEP function is provided to complement the high level interrupt structures. This statement may be used to suspend execution of the program without a "busy wait" loop until an event occurs—particularly valuable in the Amiga's multi-tasking environment.

All ROM Kernel functions may be executed from within an F-Basic program as though they were user defined subprograms. In addition, an F-Basic program is provided to facilitate the inclusion of any user defined libraries, along with those of the ROM Kernel.

F-Basic supplies an enormous variety of commands and built-in functions to perform such tasks as string pattern matching (as in SNOBOL4), conversion between strings and numeric types, the standard transcendental functions such as LN, LOG, EXP, SIN (even inverse hyperbolic trig functions!), a wide variety of graphics and animation commands, the inclusion of machine language code, direct access to processor registers, random access files, IFF picture and sound files, speech synthesis, and many more. All of these features are easily located within the manual, which contains an extensive index, as well as a special appendix listing tables of commands of similar types. The manual, which has a desktop published appearance, contains comprehensive and concise information with normally at least one programming example for each new feature introduced.

F-Basic's real arithmetic is extraordinarily fast. It possesses a specially optimized proprietary nine digit format as well as the standard IEEE double precision format. The latter may be implemented, if desired, by the F-Basic compiler as in-line coprocessor instructions instead of the slower ROM Kernel calls provided by the operating system.

In addition, the F-Basic compiler appears to perform a number of local code optimizations. For instance, logical expressions within control statements are converted into short circuit logic. At the programmer's option, the first four integer variables or the first two real variables are stored within processor registers to speed data access. Integer multiplies or divides using small integers may be optimized with an &QUICK compiler directive. Among the many compiler command options available are those which direct the compiler to generate code specific to the 68020, 68030, or 68040 (Amiga 2500, 3000, or 4000) and enhance the execution speeds on those platforms. Overall, F-Basic is competitive in speed with the best available compiled languages and is perhaps fastest in floating point of any system available on the Amiga. This fact has also been noted in previous published reviews and timing tests.

F-Basic has taken a tentative step towards object-oriented programming with the inclusion of operator overloading. That is, the user may redefine any of the single character operators to perform new operations on any user-defined record data type. The system itself employs this feature to provide built-in vector, matrix, and complex arithmetic. It is hoped that future versions will continue this trend.

F-Basic may be invoked from the CLI, WorkBench, the user's favorite text editor, or the F-Basic integrated editor environment that is packaged with the system. The latter provides a unified approach to editing, compiling, executing, linking, and debugging. After a program has been constructed from within the editor, it may be compiled without leaving the editor, and in the event of a syntax error, control automatically returns with the cursor located at the offending line. In addition, Delphi Noetic provides an optional Source Level DeBugger (SLDB) which permits the single step execution of programs, the setting of execution breakpoints, the display and modification of the values of the program variables within a fully windowed Intuition interface, and the examination and modification of memory, processor and coprocessor registers. The strength of the SLDB appears to be that it works at the source level of the program, rather than the assembly language level. Although a knowledge of machine language is not required, these enthusiasts will find that a fully featured reverse assembler is also included. The SLDB greatly simplifies and speeds up program development. A sample programs disk is also included with the system, containing over 100 examples illustrating most of the important language features.

New Features in F-Basic 5.0

Undoubtedly, the outstanding characteristic which separates the Amiga platform from competing computer hardware is the power, versatility, and flexibility of its graphics capabilities. Until 5.0, F-Basic screens were based upon the formats provided by the original graphics hardware. This meant that users with the ECS or AGA chip sets were unable to easily implement the explosion of new screen modes and window formats made available with the improved graphics hardware. With the new release, F-Basic supports an additional extended SCREEN statement, which greatly enlarges the choices available. The most important of these allows the user to select from a palette of the over 100 total screen modes (depending upon the user's monitor, chip hardware, and AmigaDOS version) supported by the operating system. These new modes significantly enhance the screen resolution, depth and number of colors available per screen, and "look" available to the F-Basic programmer.

In addition to the support for the ECS and AGA chip sets, each of the four Overscan types (Text, Standard, Max, and Video) are available with F-Basic screens and windows. For those unfamiliar with overscan, the effect is to permit larger portions of the display screen to be accessible for the rendering of text or graphics. Alternatively, one may specify that the screen opened match exactly the WorkBench screen, saving the programmer some effort. Finally, new F-Basic 5.0 screens support PAL formats for European users of the system. These formats typically have more scan lines per display and a higher refresh rate than the corresponding NTSC or Multi-Scan monitors. Compared with the four limited screen modes available in previous versions of F-Basic, 5.0 opens whole new vistas on graphical interface programming.

Continuing the theme of enhancing the graphical interface capabilities of F-Basic, Version 5.0 provides an easy sequence of high level commands for installing new fonts. These permit printing with different text fonts and colors within the same window among other things.

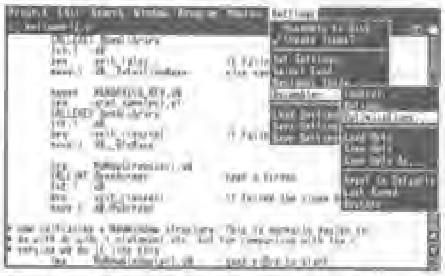
Earlier versions of F-Basic printed text in windows using the Amiga's console device that limited text lines to locations that were an even multiple of the font height in pixels. To provide further flexibility in the construction of displays, Version 5.0 uses an alternate print function that allows text to be

Express Yourself

with Languages & Samplers from Oregon Research!

Devpac 3 **$149.95**

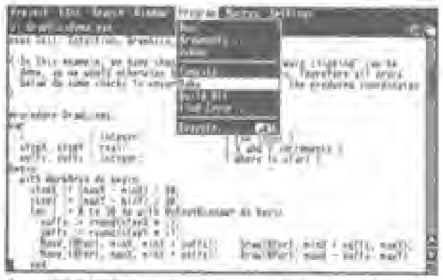
With it's powerful multi-windowed integrated editor/assembler/debugging development system, Devpac 3 is the ideal programming environment for beginners and professionals alike.



The heart of the package is the fast and powerful assembler and debugger supporting all 680x0 processors and coprocessors which is now at least 40% quicker than its predecessor. Compatible with all Amiga computers and includes specific library support for Workbench 1.3, 2, and 3.

HighSpeed Pascal **$199.95**

The leading Pascal development system for all Amiga computers. Compatible with Turbo Pascal 5.0 on the PC, the system includes an integrated multi-window editor and interactive error detection and a compiler that processes more

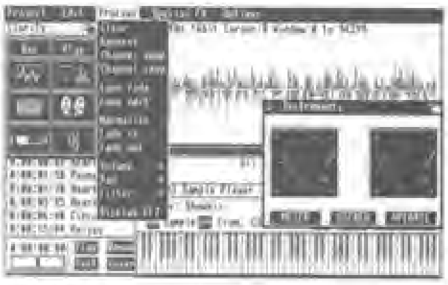


than 20,000 lines per minute. Also supplied is a stand-alone CLI compiler, inline assembler for ultimate speed, and versatile make facility for easy project management. These features make HighSpeed Pascal a truly powerful and easy-to-use system for all levels. Compatible with all Amiga computers and includes specific library support for Workbench 1.3, 2, and 3.

Clarity 16 **$289.95**

The first low cost professional 16 bit stereo sound sampler for the Amiga range of computers. The system can record 8 or 16 bit samples at up to 44.1 KHz from any sound source and playback to any amplifier or mixer. Also included is a complete MIDI interface for use with any MIDI instrument and commercial MIDI software applications.

The software package includes a powerful multitasking windowed sample editor with advanced editing and signal processing capabilities. The system can also perform real time effects processing as well as function as a MIDI sample sequencer. Clarity 16 is compatible with all Amiga computers including the A1200 and A4000.



And More...

Also from Oregon Research, **HiSoft Basic 2** a professional BASIC development system; **Power Basic** an entry level structured BASIC; **P.F.M.+** a powerful personal and small business financial management system; **AMAS2** and **StereoMaster** professional and entry level 8 bit stereo sound samplers with integrated MIDI; **MegaLoSound** 8 bit sampler with direct to disk recording; and **ProFlight** an amazing Tornado flight simulator

OREGON RESEARCH

16200 S.W. Pacific Hwy., Suite 162
Tigard, OR 97224
PH:(503) 620-4919 FAX:(503) 624-2940

positioned beginning at any pixel location within the window. In addition, new commands sense the current width and height of the active font, as well the number of font characters per line and the number of lines per window. This aids in printing text and graphics to a window whose size may be arbitrarily changed by the user during execution.

The repertoire of high level interrupts has been enhanced with the inclusion of a window re-size event and a serial port event. The use of high level events in 5.0 is illustrated in Listing 3, where both the syntax of an event block and a typical example of its use are shown.

The last several versions of F-Basic have had the capability of transmitting data over the serial port using its PRINT# command. However, to receive data using the INPUT# statement meant that the F-Basic program would suspend execution when this command was encountered until some new data arrived at the serial port. One could, of course, always get around this limitation by using the ROM Kernel facilities that can be accessed through F-Basic, but this again required that the programmer become familiar with the internals of the ROM Kernel data structures. Listing 4 illustrates the new SERIAL event block and the simplicity with which the latest serial information is accessed. In addition, 5.0 adds high level commands that may be used to alter the characteristics of the serial port, i.e. baud rate, parity, error checking, etc.

The Version 5.0 editor and SLDB, which along with the compiler are themselves written in F-Basic, have apparently taken advantage of many of the new improvements in the graphical interface. The earlier F-Basic editor was restricted to operating within a limited screen format while the new editor is capable of adjusting to any of the various screen modes and fonts available on the Amiga. It therefore possesses the new 3-D look when invoked within the newer operating systems. This seems to provide a more pleasant "feel" as one uses the integrated environment.

Although space permits touching on a just a few major innovations in 5.0, a number of incremental changes and code fixes have also been added by DNS.

Conclusion

F-Basic 5.0 is a language system that should be considered by any serious Amiga developer, or by a beginner or intermediate programmer who wishes the least painful transition from standard BASIC to a more modern language. As the step from the ease of BASIC to the powers of C has been best described as treacherous, F-Basic seems to bridge that gap in an intuitive and understandable way. F-Basic 5.0 is a powerful, comprehensive, modestly priced program-development system and is one of only a handful of programming languages for the Amiga still under active support and continued development. ✓

F-Basic
Delphi Noetic Systems
2700 West Main St.
Rapid City, SD 57709
(605) 348-0791

LISTING #1

```
PROGRAM AnnualInterest

DOUBLE
Principal,MonthlyInterest,TotalInterest,Payment,InterestRate
INTEGER Month

' REM This program computes the amount of each monthly
payment
' which goes to principal and to interest as well as the
total
' interest paid during the year

Principal=10000
Payment=[illegible]
TotalInterest=0
InterestRate=0.[illegible]

FOR Month=1 TO 12
  MonthlyInterest=(Principal*InterestRate)/[illegible]
  TotalInterest=TotalInterest+MonthlyInterest
  Principal=Principal-Payment+MonthlyInterest
  PRINT "The Interest Paid In Month",Month,"
is:",MonthlyInterest
NEXT Month

PRINT "The Total Interest For The Year is:";TotalInterest
END
```

The LANGUAGE For The Amiga!

One Amiga language has stood the test of time.

This new package represents the fourth major upgraded release of F-Basic since 1988. Packed with new features, 5.0 is the fastest and fullest yet. The power of C with the friendliness of BASIC. Compatibility with all Amiga platforms through the 4000...compiled assembly object code with incredible execution times... features from all modern languages, an AREXX port, PAL and ECS/AGA chip set support...Free technical support... This is the FAST one you've read so much about!

F-BASIC 5.0™

Supports DOS 1.3, 2.0, 2.1 and 3.0

F-BASIC 5.0™ System $99.95
Includes Compiler, Linker, Integrated Editor Environment, User's Manual, & Sample Programs Disk.

F-BASIC 5.0™ + SLDB System $159.95
As above with Complete Source Level DeBugger

Available Only From: DELPHI NOETIC SYSTEMS, INC. **(605) 348-0791**
P.O. Box 7722 Rapid City, SD 57709-7722
Send Check or Money Order or Write For Info. Call With Credit Card or C.O.D.
Fax (605) 342-2247 Overseas Distributor Inquiries Welcome

LISTING #2

```
PROGRAM [illegible]

' [illegible]
[illegible]

GLOBAL
  INTEGER [illegible]
  TEXT [illegible]
maximum [illegible]
  INTEGER [illegible]
```

AC's TECH

wants you!
We are looking for Amiga programmers and developers to write articles for the magazine.
If you are

Interested

send for a set of our author guidelines.
For more information

Call

1-800-345-3360

```
 ; The above Variables are all GLOBAL and are accessible
by every module.

SUBPROGRAM
 SUBROUTINE JUMBLE

INPUT "Enter String To Jumble (Maximum 8 Characters):";LINE
N=LENGTH(LINE)
NUM=0
L=0
TEMP=" "              ;
OLD=" "          ; ? These text strings are blank filled
USED=0             ; ? USED is a boolean value which
remembers
                          ; ? whether a letter has
already been used
CALL JUMBLE()
PRINT "Found ",NUM," Distinct JUMBLE Combinations"
END

SUBROUTINE JUMBLE
LOCAL
  INTEGER I
FOR I=1 TO N                  ; ? For each letter in the given
word
   IF NOT USED(I) THEN ; ? Has the I-th letter been used?
      USED(I)=1
      L=L+1
     TEMP(L)=LINE(I)  ; ? If not, use it
     CALL JUMBLE( )   ; ? Recursive call which fills in
the
                                   ; ? remaining letters in
the current "word"
     USED(I)=0
    IF L=N THEN ; Have we used all N letters?
       IF NOT (TEMP IN OLD) THEN    ; ? the IN operator
searches the
                                      ; ? list OLD to see
if the current
                                       ; ? word has
already been generated
          NUM=NUM+1
         OLD(NUM)=TEMP ; ?  Store new combination in
array OLD
          PRINT TEMP   ; ?  and print it out
        ENDIF
     ENDIF
     L=L-1
  ENDIF
NEXT I
RETURN
END
```

LISTING #3

"THIS LISTING IS NOT COMPLETE AS IT APPEARS"

```
PROGRAM WindowResize

<declarations here>
<window open statements here>

ON WINDOW_RESIZE EVENT
  N=WINDOW_NUMBER
   Temp=$MAX_YMAX #N ( )
  LinesPerWindow=HIWORD(Temp)
  CharsPerLine=LOWORD(Temp)
  Width=WINDOW_INFO(3)
  Height=WINDOW_INFO(4)
ENDEVENT

<balance of program>
END
```

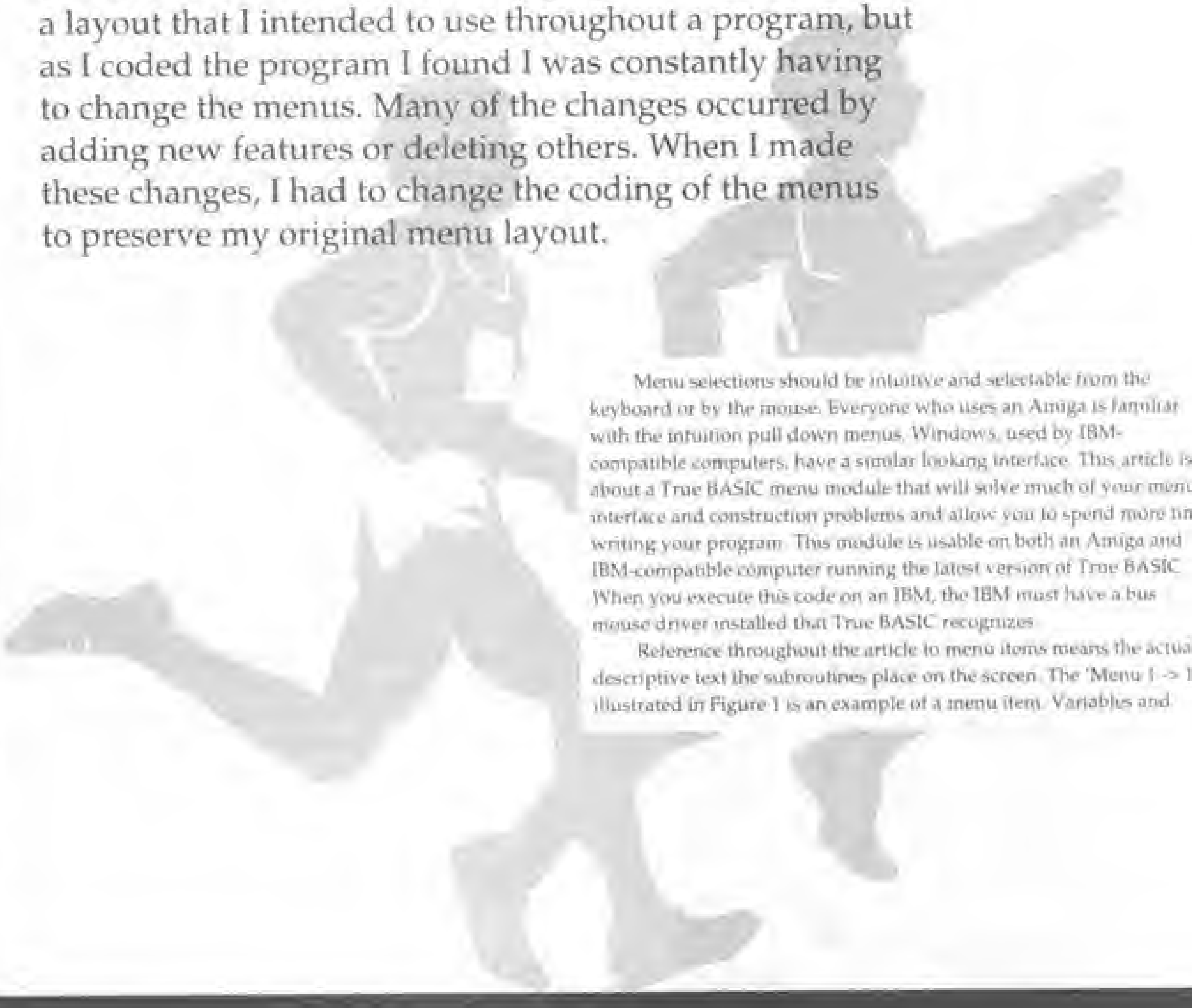
Complete listings and source code can be found on the AC's TECH disk.

Please write to:
Jeffrey Stein
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722

Quick Menus for True BASIC Programs

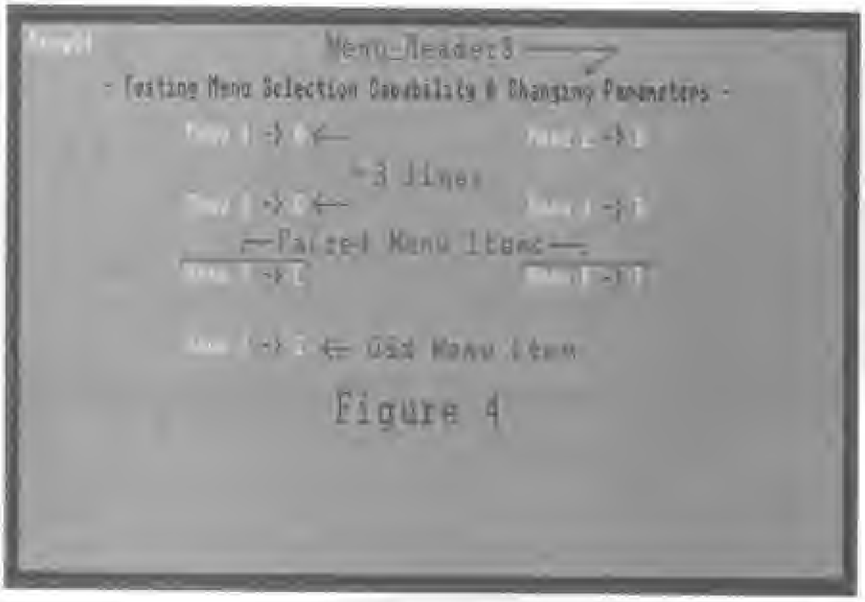
by T. Darrel Westbrook

Writing a user menu for use throughout a program can consume much time. For my projects I would design a layout that I intended to use throughout a program, but as I coded the program I found I was constantly having to change the menus. Many of the changes occurred by adding new features or deleting others. When I made these changes, I had to change the coding of the menus to preserve my original menu layout.



Menu selections should be intuitive and selectable from the keyboard or by the mouse. Everyone who uses an Amiga is familiar with the intuition pull down menus. Windows, used by IBM-compatible computers, have a similar looking interface. This article is about a True BASIC menu module that will solve much of your menu interface and construction problems and allow you to spend more time writing your program. This module is usable on both an Amiga and IBM-compatible computer running the latest version of True BASIC. When you execute this code on an IBM, the IBM must have a bus mouse driver installed that True BASIC recognizes.

Reference throughout the article to menu items means the actual descriptive text the subroutines place on the screen. The 'Menu 1 -> 1' illustrated in Figure 1 is an example of a menu item. Variables and



subroutines are in italics and capitalized words are keywords. Line numbers are for reference only.

I used the following guidelines to write the module:

- Menus would be self centering (horizontally and vertically).
- Use module SHAREd variables to reduce passing variables to and from subroutines.
- Menu items would be selectable from the keyboard (single character) and by using the mouse by clicking on a menu selection.
- The returned value from the menu selection subroutines should be a logical and consistent format and not be dependent on the menu items.
- The programmer supplies the menu items and the single characters used by the menu subroutines.
- Menu construction and display should be reasonably fast.

There are two menu types in the Menu Module. The first menu type displays a two-column menu with a maximum of 46 display selections. I'll refer to this as **Menu Column** type. The second menu type is a free flow display. It has limits, but they are dependent on several factors, such as menu item length, unique single keyboard input, lateral separation between menu items, and spacing between display rows. I'll refer to this as a **Linear Display** type. Before I get to

the details of how each menu type operates, I should briefly explain the Global Module and then explain some of the variables used in the Menu Module.

The Global Module is small. When you incorporate the Menu Module into your programs, ensure that you include lines 106 through 108. Following the Global Module is the Menu Module.

Menu Module variables, that you assign values, follow:

Menu_Header is the directions or instructions for the user. In my examples, it is the text string, "- Select by key or use mouse -".
Menu_Space is the number of lines between menus. The actual spacing between the displayed menu items is (Menu_Space - 1). The minimum is one, which results in no spacing between displayed menu items.
Menu_Pen_Color is the menu item text color.
Menu_Arrow_Color is the color of the arrow used in the **Menu Column** type (Figure 2) and is the character highlight color for the **Linear Display** type (Figure 5).
Msg_Pen_Color is the color used for the *Menu_Header*
gap is the minimum number of horizontal spaces between the **Linear Display** menu items.

You can change these variables at any time by calling the Menu Module *Change_Menu_SHARE* subroutine. These variables retain their respective values within the module until you change them. This is a characteristic of the module specific SHARE statement (see lines 115 to 119). Since I used the SHARE keyword, I did not need to devise ways to keep these values, pass them to calling subroutines, or recalculate them. *The True BASIC Reference Manual*, copyrighted 1988, states that SHAREd values become STATIC global variables within the confines of the module. This is a descriptive and logical view of how SHAREd variables behave within a module.

After the user selects a menu item, the *Erase_Menu* subroutine erases the menu (see last subroutine in Menu Module). This subroutine erases only the lines that the previous menu subroutines placed on the screen. It does not clear the whole display. This allows you to place whatever text you want on the screen and the Menu Module will not erase it.

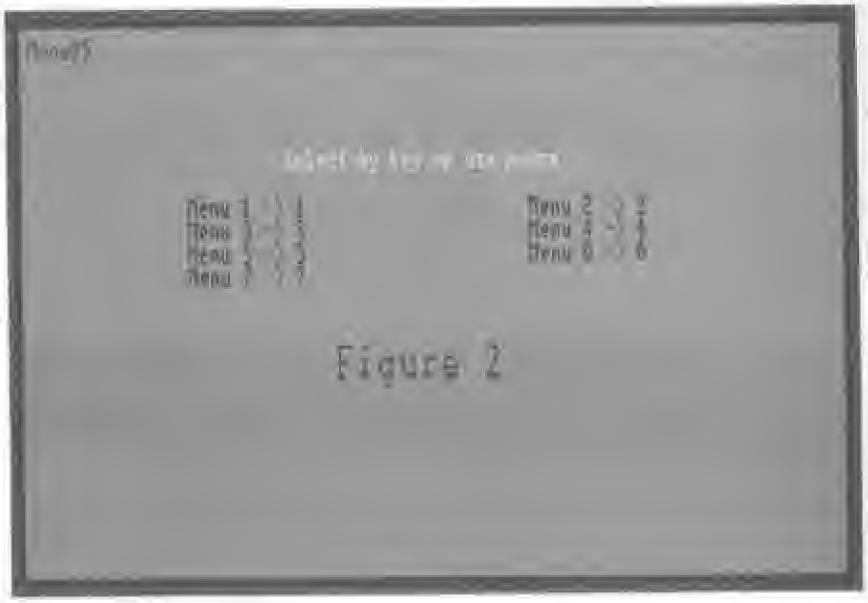
There are six subroutines within the Menu Module. I have discussed two of them. The remaining four set up and control selections for the **Menu Column** and **Linear Display** type menus. I'll discuss the mechanics behind the **Menu Column** type and then finish with the **Linear Display** type.

The following information may help you get a pictorial view of the **Menu Column** variables as I discuss them and their usage.

```
Menu 1 -> 1
Menu 2 -> 2  <- Menu_Min_Row
Menu        3 -> 3
Menu 4 -> 4
Menu_Size is 6
Menu_Space

Menu 5 -> 5
Menu 6 -> 6  <- Menu_Max_Row
Menu 7 -> 7
Menu_Odd is one
```

Menu Column type uses the *Display_Menu* and *Display_Menu_Select* subroutines. Paired menu items (i.e., a left menu column and a right menu column) are the foundation of this menu set up. The *Menu_List* array stores the menu item text and is used to pass information to the subroutine *Display_Menu*. Program lines two through eight is an example of passing this information. The odd array elements (like *Item_Selected$* (n,1) ... (n,3), etc.) are the left menu column and the even array elements (like *Item_Selected$* 4(n,2) ... (n,4), etc.) are the right menu column.

Menu_Elements variable controls the screen display of *Menu_List*. *Menu_Elements* represents the total menu items and is the character length of the letters variable. Using this approach, I did not need to check each element of *Menu_List$* array for a null value. When the last even element in *Menu_List* is null, then the subroutine will not attempt to display that element on the right side of the screen. Null menu items elsewhere in the *Menu_List* array are printed (or not printed, depending on your viewpoint) to the screen.



Line five represents an odd number of menu items with the last item being a null value. *Menu_Elements* is equal to seven which is the number of characters in the letters variable. To understand how this works, delete the number seven from the letters$ variable on line eight, then run the program. The last option, 'Menu 7 -> 7', will not appear nor is it keyboard- or mouse-selectable. This characteristic prevents the *Item_Selected$*(4,2) null string from being selectable as a valid menu item.

Menu_Const (menu constant) is the number of menu item matched pairs. *Menu_Const* is the integer of length of letters$ divided by two. When letters is seven characters in length, then *Menu_Const* equals three. I used the *Menu_Odd* variable to flag when an odd number of menu items is in *Menu_List*. When *Menu_Odd* is one, then an odd number of menu items exist. The subroutine uses this information to calculate the number of display rows needed for the menu. Once the subroutine determines the number of screen rows, it then calculates *Menu_Min_Row* and *Menu_Max_Row* (see equations, lines 135 and 140).

Menu_Max_Row is the last row of paired menu items. When *Menu_Odd* is equal to one, the subroutine places the odd menu item on the next display row (*Menu_Max_Row* + *Menu_Space*).

The *Display_Menu* subroutine checks *Menu_Min_Row* to determine if the menu items will fit laterally on the screen. It does this by ensuring that (Menu_Min_Row - 2 > 0). When Menu_Min_Row - 2 is equal to or less than zero, the subroutine invokes a runtime error and the program stops. It flags the problem with the message "Minimum row is less than one." (see lines 137 to 139).

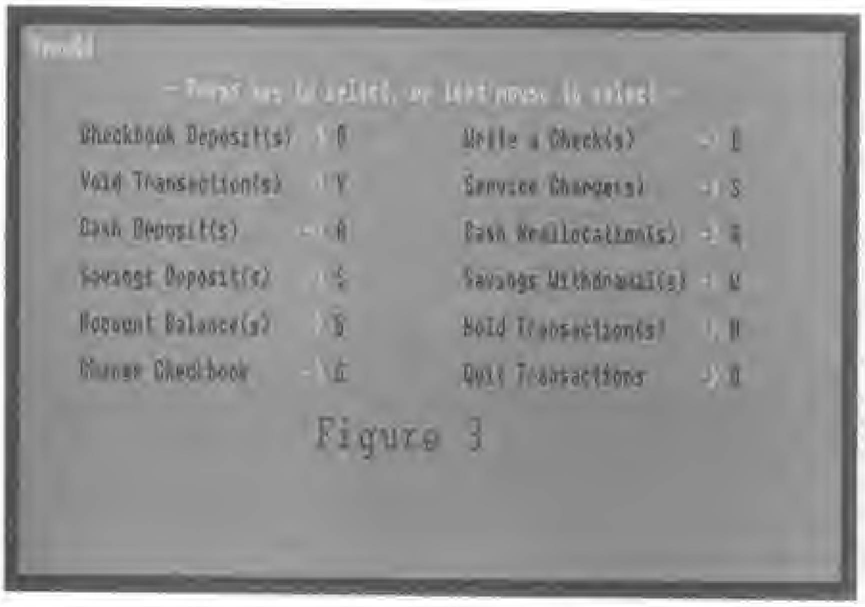
Once the subroutine calculates Menu_Max_Row, it determines horizontal spacing by finding the length of the largest left and right menu item (see lines 141 through 145). The equations used to calculate the lateral separation is simple, but several of the constants in the equations may not be obvious (reference lines 149 to 153). The number five in the equation is the number of characters needed past the Left_Len and Right_Len, to allow for blanks and the right pointing arrow for each menu item. The minus one keeps the column addition correct. Menu items have a minimum of four spaces between left and right menu columns (line 152). Once the program flow determines all these variables, it places the menu on the screen (lines 154 to 187).

The remaining code of Display_Menu loads the array with a three character string which is the ASCII ordinal code for each of the letters characters. For example, if *A* is one of the letters$ characters, then the three string character loaded into s$ is *065*. I used this approach rather than numeric value to speed up selection operations.

The Menu Module subroutines return *one* variable, *Item_Selected*, in the form of *MenuXY* where *XY* is the menu sequence number (like 01, 02, 03, etc.). For example, if the user selected the fourth menu item then *Item_Selected* would be *Menu04*. The programmer needs a simple CASE SELECT structure to control program flow after a menu selection. The CASE SELECT structure could look like:

```
SELECT CASE Item_Selected$
CASE "Menu01" ! first menu item selected
(program statements)
CASE "Menu02" ! second menu item selected
(program statements)
CASE "MenuXY" ! the last menu item selectable
(program statements)
END SELECT
```

The module subroutines return only valid menu selections so



```
[illegible]

      - [illegible] key to select, or left mouse to select -
  Checkbook Deposit(s)   -> [illegible]     Write a Check(s)        -> [illegible]
  Void Transaction(s)    -> [illegible]     Service Charge(s)       -> S
  Cash Deposit(s)        -> [illegible]     Cash Reallocation(s)    -> [illegible]
  Savings Deposit(s)     -> [illegible]     Savings Withdrawal(s)   -> [illegible]
  Account Balance(s)     -> [illegible]     Hold Transaction(s)     -> [illegible]
  Change Checkbook       -> [illegible]     Quit Transactions       -> [illegible]

                     Figure 3
```

[illegible] and lets you concentrate on content and style and not worry about the meticulous details of getting your selections displayed. Once you understand the Menu Module operation, you can change the subroutines or add other subroutines for even more powerful menu selection capabilities.

The hard part is done. *Display_Menu_Select*, the second subroutine of the **Menu Column** type, handles the menu selection process. The CALL Buffer statement clears previous keyboard and mouse inputs. Program flow then loops until the user selects a menu item with the keyboard or by clicking on an item with the left mouse button. The keyboard menu selection converts lower case to upper case (line 208) and then compares it to a three character string in the array. When the user selects a valid menu item, the array element becomes the menu item selected. If you press the letter A and x (3) equals 065, then *Item_Selected* is Menu03.

Mouse menu is only valid when you release the left mouse button while the mouse pointer is over one of the menu items. The Display_Menu_Select... subroutine determines the row and column of the mouse click (lines 222 and 223). It then compares the information to the known rows and columns of your menu items. When it finds a match, it returns *Item_Selected* to the calling routine.

The determination of *Item_Selected* in lines 228 to 247 may confuse you. The equations of line 231 and 239 determine *Item_Selected* for all occurrences of Menu_Space, except when it is equal to one. I added the capabilities of lines 229 and 237 so the subroutine could handle the special case of Menu_Space equal one.

Between my explanations and your study of the module code, you should have a good understanding of how Display_Menu and Display_Menu_Select subroutines operate. Understanding the **Menu Column** type is essential for understanding the **Linear Display** type.

I call it a **Linear Display** type because I create one long string of the menu items, then I take it apart in such a manner that it would fit on the screen. The biggest difference between the **Linear Display** and Menu Columns is that keyboard selection is now case sensitive. The following is a summation the process.

- Load *Item_Selected* items into menu(n,1).
- Load individual letters [illegible] into menu(n,2).
- Load the ordinal position of letters[illegible] into menu(n,3). For a [illegible] *Manual*, page 231, or the *Amiga Student Edition Manual*, page 175.
- Build a string with menu items separated by *gap* number of spaces.
- Take the string apart based on the lateral screen size (c_max), but keep menu items intact.
- Load each menu display row into the array r.
- Determine the start and end column of each menu item.
- Create a six character string composed of row, start column, and end column and load it into menu(n,4). For a menu item on row 9, column 10 to 21, this value would be 091021.
- Display the menu and highlight the letters that allow keyboard selection.

The last example in the program listing uses the **Linear Display** type. The data for *Item_Selected* is in the subroutine Load_Menu_1. Notice the menu items' alignment. I added spaces to some of the menu items so they would all align in three columns on the screen. The subroutine works properly without these spaces, but the spaces made the display look better. The *gap* variable establishes the minimum spacing between each column. The preceding menu item column determines where you must place spaces to align each column. The longest menu item in the first column is "Checkbook Deposit(s)". Align all the menu items in the second column with "Write a Check(s)", which is on the same row as "Checkbook Deposit(s)." You cannot reduce the spacing between "Checkbook Deposit(s) and "Write a Check(s)" so you increase it for the other second column menu items.

The second column menu item "Savings Withdrawal(s)" is the basis for the third column spacing. Therefore, align the third column menu items with "Account Balance(s)" by adding spaces to the other menu items. The extra spaces have no effect on the placement of the highlighted menu item letters. The number of spaces in the variable *gap* has to be one space larger than the largest number of consecutive spaces within your menu items. If you have a two word menu item and you use three spaces to separate the two words, then *gap* must be

OWN AN AMIGA 1200 TWICE THE SPEED OF AN A4000/030 OR A3000 SYSTEM!
GET TRUE 32-bit WIDE, ZERO WAIT-STATE FASTRAM ACCESS!
"PURCHASE THE MICROBOTICS M1230XA CARD FOR THE AMIGA 1200"

AMIGA

A4000 Computer	CALL
A1200 Computer	CALL
A600 Computer	175
1942 Multisync Monitor	399
1084S Monitor	249
A2630 Accelerator w/4MB	449
A520 Video Adapter	34
A2088 XT Bridgecard	69
A2091 Hard Drive & Ram Controller w/120MB HD	269
A2091 HD Controller	65
2 MB Ram For 2091	80
Janus 2.1 Update	35
A2000/3000 Disk Drives	69.95
A2000/3000 Keyboard	59.95
A2000/3000 Power Supply	109
A2000 Bigfoot 300W PS	188
A500 Bigfoot PS	89
External case PS & Cables	125

A2320 Flicker Fixer
Only $129

AMIGA CUSTOM CHIPS

Kickstart 2.1 Upgrade Kit	77.95
2.04 Rom (HD Drives)	33.95
Kickstart 1.3	21.95
1MB Agnus (8372A)	37.50
2MB Agnus (8372B)	79.95
Super Denise (8373)	28.95
Paula (8364) Or Denise	18.95
CIA (8520)	9.50
Gary (5719)	13.95
2620/2630 Upgrade Kit	35
2091 Upgrade Eproms 7 +	35

IVS

Grand Slam	229
Grand Slam 500	267
Trumpcard Pro	139
Trumpcard 500 Pro	225
Trumpcard 500 Plus	149
Trumpcard 500 AT	164
Sourcer Switching Power Supply	99
Moviemaker	799

VIDEO

DPS Personal TBC III	875
DPS Personal TBC IV	799
DPS Personal Animation	1599
DPS Personal Component	399
DPS Personal V Scope	699
Kitchen Sync	1199
Opal Vision 2.0	845
DCTV-NTSC	274
Retina w/2MB	479
Retina w/4MB	549
Vlab-24Bit Digitizer	379
Scala 210MM	209
Deluxe Paint IV AGA	119
Morph Plus	149
Art Department Pro	159
Pixel 3D Professional	149
Video Director	129
Caligari 24	239
Aladdin 4D	249
Video Toaster 3.0 Upgrade	849

ICD

AdIDE 40 MB HD system for Amiga 500 System	220
AdIDE 80 MB HD System for Amiga 500 System	280
Flicker Free Video II	228

DKB

Insider II w/1.5M RAM	180
2632 w/4Megabytes	349
MegAChip 2000/500 w/2MB Agnus	179
Multi-Start 2 Rev 6A	29
KwikStart II for A1000	69
SecurKey Security Board	99
BattDisk battery backed static RAM disk	189

MICROBOTICS

M1230XA ACCELERATOR
68030 RC25Mhz CPU
w/MMU & clock **Only $249**
50 MHZ VERSION FOR ONLY 349

MBX 1200z COMBO BOARD
68881 RC20Mhz FPU & with clock
NOW ONLY $125

Different CPU, FPU & Ram Located On Far Right Column

VIDEO TOASTER 4000 SYSTEMS

VT4000 SYSTEM #1
A4000 Computer System
10 MB Of 32 Bit Ram
120 MB Hard Disk Space
Newtek Video Toaster 4000
1084S RGB Stereo Monitor
Basic System Only $4549

VT4000 SYSTEM #2
A4000 Computer System
18 MB Of 32 Bit Ram
460 MB Hard Disk Space
Newtek Video Toaster 4000
1084S RGB Stereo Monitor
Advanced System Only $5349

VXL * 30 ACCELERATOR

68030 & 32 Bit Wide Ram For The A500/2000

VXL 30 25 & MMU Mhz $185 w/68882 25FPU$235 | 40EC Mhz $275 w/FPU $390 | 33 Mhz & MMU $299 w/FPU $385 | VXL 32 RAM 2MB $219 8MB $589

Great Valley Products

A500-HD8+0MB/52Q	275
A500-HD8+0MB/85Q	329
A500-HD8+0MB/120M	379
A500-HD8+0MB/245Q	435
A530-HD8+1/120M	545
A530-HD8+1/245Q	649
A2000-HC8+0MB	149
SIMM32/1MB/60ns	69.95
SIMM32/4MB/60ns	CALL
1MB SIMM GForce A3000	179
G-Lock Genlock	385
A3000-Impact Vision 24	1199
A2000-IV24 Adapter	55
VIU-CT	499

A1230 Turbo+ 40/40/4	$549.00
A1200 SCSI / RAM+	$275.00

GForce Accelerators w/ 68030,68882, 40ns RAM, SCSI Controller, RAM Card in One

40Mhz/4MB	575
68040 33Mhz A2000/4MB	975

GVP 68030/ 68882-50Mz 4MB/60ns RAM w/SCSI Controller $795

PC286 Module 16Mhz	59
Tahiti-II 1GB (35ms)	3100
Tahiti-II 1GB Cartridge	289
Syquest 44MB Removable	259
Image F/X Upgrade	29
Syquest 88MB Removable	365
Impact Vision 2.0 Update	119
68882 40Mhz FPU PLCC	129
FaaastROM Kit (For HDs)	35
Cinemorph Software	55
Phonepak VFX	289
DSS8+ Sound Sampler	99
I/O Extender (2SerialPort)	99
Image F/X	199

AMIGA 4000 & 3000 OWNERS CRUSH THE 16 MEG BARRIER
- Add up to 128 MB of ram of contiguous memory
- Four simm sockets using industry standard simms
- Create and run animation from ram on the VT4000
- True Zorro III 32 bit memory board
- Play back over 50 seconds of real-time animation in Hi-Res
- Record your animation at a fraction of the cost!

DKB's 3128 is Only $395

Video Toaster 4000 Systems 3449.00
We can configure a system to suit your requirements.

ASK ABOUT OUR ACCELERATOR, HARDDRIVE AND MEMORY UPGRADES

UNMATCHED SYQUEST PRICES

44MB DRIVE (SQ555)	$249
88MB (SQ5110C) (R&W44)	$349
105MB IDE	$449
105MB SCSI	$525
44MB Cartriges	$ 65
105MB Cartriges	$ 98
External Versions Add	$ 99

A2386 Bridgecard Owners Special Upgrade Package
- Includes 80387-25 Mhz Floating Point Unit
- Janus 2.1 Updated Software
- 4 1x4 Page Zip Chips For 2 MB Additional Ram

Only $164

VECTOR
THIRD GENERATION 68030 PROCESSOR ACCELERATOR FOR THE AMIGA 2000
$529

HARD DRIVES

Warrenties
Quantum=2 Years
Maxtor=1 Year

Maxtor 120 LPS SCSI/IDE	199
Maxtor 170 LPS IDE	229
Maxtor 213 LPS SCSI/IDE	265
Maxtor 245 LPS SCSI/IDE	285
Maxtor 346 IDE	365
Quantum 85 (Low Profile)	185
Quantum 105 PDrive HH	199
Quantum 127 ELS	209
Quantum 245 LPS SCSI-2	295
Quantum 525 LPS 10MS	699
Quantum 1.2 Gig 5 YR	1199
Toshiba 1.2 Gig 5 YR	1199

A1200/600 Hard Drives

85M Conner / Seagate	$215
130M Conner / Maxtor	$315
235MB Seagate / Toshiba	$449

MEMORY CHIPS

All Speeds Available Call If Not Here

1x8 100-60ns SIMMS	32.00
4x8 80-60ns SIMMS	129.00
1x4 80-80ns Static ZIP	18.50
1x4 80-80ns Page ZIP	17.50
1x4 80-50ns Page DIP	19.95
1x1 120-70ns DIP	3.25
256X4 120-80ns DIP	4.00
256X4 120-80ns ZIP	4.00
256x32 (1MB Simm 72Pin)	29.00
512x32 (2MB Simm 72Pin)	75.00
1x32 Simms (Microbotics,Etc)	139.00
2x32 Simms (All Speeds)	299.00
4x32 Simms (All Speeds)	599.00
8x32 Simms (All Speeds)	CALL

Please Call Before Ordering, Ram Changes For Better Or Worse

MATH CHIPS, CPU's &FPU's

68030-RC-50 w/MMU	149.00
68882-RC-50 (PGA)	136.95
68030-RC-33 w/MMU	129.00
68882-RC-33 (PGA)	95.00
68030-RC-25 w/MMU	89.00
68882-RC-25 (PGA)	75.00
68030-FN-PLCC (Various)	CALL
68882-FN-PLCC (Various)	CALL
80387-25SX (Bridgecards)	89.95
Crystal Oscillators (All)	10.00

A600/1200 UPGRADES

PCMCIA 2 and 4MB	129/209
Baseboard 601&601C	54/59

LASER PRINTER MEMORY

HP II, IID, IIP, III, IIID, IIP AND ALL PLUS SERIES

Board with 2MB	79.00
Board with 4MB	145.00
Deskjet 256K Upgrade	55
HP 4 (4 Meg)	149
HP 4 (8 Meg)	295

AUDIO

Sunrise Industries

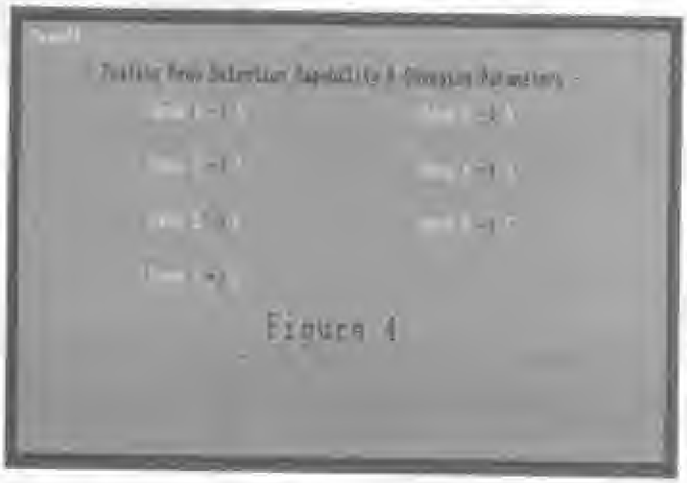
AD516 Digitizer (16Bit)	1189
AD1012 Digitizer (12Bit)	479

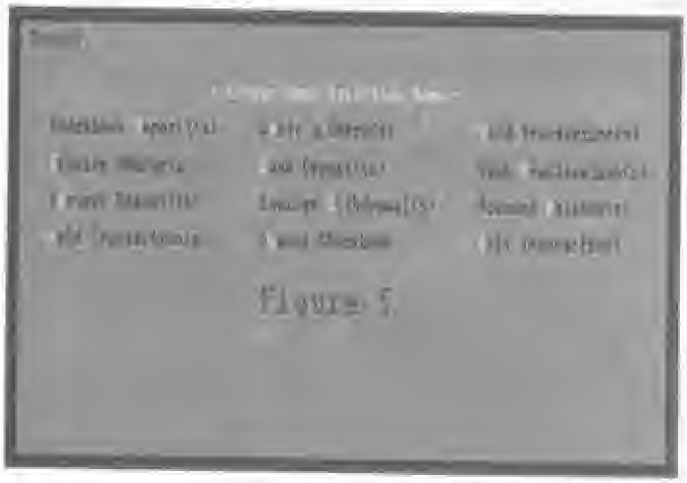
ACCESORIES/MISC.

PowerPlayers Joystick	6.49
CSA DERRINGER 25/25	399
CSA DERRINGER 50/4	699
CSA Rocket Launcher	499
SupraTurbo 28Mhz	149
Safeskin Protectors	15.00
Xtractor+ Chip Puller	9.95
Slingshot single A2000 slot for the A500	39.95
Slingshot Pro	69.95
Kool-It Cooling kit A500	39.95
Qwika Switcha 4 socketed ROM selector	39.95
Power Connectors	CALL
SCSI HD Cables	CALL

DeVine COMPUTER SALES
18 Wellington Drive
Newark, DE. 19702
(800) 578-7617 ORDERS ONLY
(302) 836-4138 PRODUCT Info/Tech
(302) 836-8829 Fax 24 HOURS

Please Understand Our Policies
VISA / MASTER Card Accepted. Prices And Specifications Are Subject To Change Without Notice! 15% Restocking Fee On All Returns. Defective Merchandise Will Be Replaced With Same Item. Call 302.836.4145 For Approval RMA# Before Returning Merchandise Merchandise. No Returns After 10 Days From Delivery Date Not Responsible For Incompatibility Of Products Shipping And Handling For Chips Is $5 COD Fee $6 Personal Checks Require 10 Working Days To Clear. Call For Actual Shipping Prices On All Other Items. Ram Prices Change Week To Week





- Menus would be self centering (horizontally and vertically).
- Use module SHAREd variables to reduce passing variables to and from subroutines.
- Menu items would be selectable from the keyboard (single character) and by using the mouse by clicking on a menu selection.
- The returned value from the menu selection subroutines should be a logical and consistent format and not be dependent on the menu items.
- The programmer supplies the menu items and the single-characters used by the menu subroutines.
- Menu construction and display should be reasonably fast.

```
!
! Portable True BASIC Menu Module
! Copyright 1993 by T. Darrel Westbrook
!
DECLARE PUBLIC c_max
DATA "Menu 1","Menu 2"
DATA "Menu 3","Menu 4"
DATA "Menu 5","Menu 6"
DATA "Menu 7",""
DIM Menu_List$(4,2)
MAT READ Menu_List$
LET Letters$ = "1234567"
!
DO                                    ! forever loop
  CALL Display_Menu(Menu_List$,Letters$)
  CALL Display_Menu_Select(Selected$)
   SET color 1
   SET cursor 1,1
   PRINT Selected$
   IF selected$ = "Menu 7" then EXIT DO
LOOP
CALL Erase_Menu                       ! clear only the current
menu

DATA "Checkbook Deposit(s)"   ,"Write a Check(s)"
DATA "Void Transaction(s)"    ,"Service Charge(s)"
DATA "Cash Deposit(s)"        ,"Cash Reallocation(s)"
DATA "Savings Deposit(s)"     ,"Savings Withdrawal(s)"
DATA "Account Balance(s)"     ,"Hold Transaction(s)"
DATA "Change Checkbook"       ,"Quit Transactions"
MAT Menu_List$ = nul$(6,2)
MAT READ Menu_List$
LET Letters$ = "DwVSCRSwBHCQ"     ! example of mixed
upper/lower case
LET MS$ = " - Press Key to select, or left mouse to select
-"
CALL Change_Menu_SHARE(MS$,2,3,5,5,c_max,9)
DO
  CALL Display_Menu(Menu_List$,Letters$)
  CALL Display_Menu_Select(Selected$)
   SET color 5
   SET cursor 1,1
   PRINT Selected$
   IF selected$ = "Menu 2" then EXIT DO
```

at least four spaces.

To demonstrate the Menu Module use, I've included four representative menus. Figures 2 through 5 are examples of the menu displays. In each of the figures, the upper left hand corner represents the returned *Item_Selected* variable. The displayed *Item_Selected* is a visual feedback of the menu item selected. I provided several different methods of loading the *Item_Selected* array for your review. The Menu Module will save time and effort when you design menu selections for your programs written in True BASIC.

```
LOOP
CALL Erase_Menu

CALL Load_Menu(Menu_List$,letters$)
DO
  CALL Display_Menu(Menu_List$,letters$)
  CALL Display_Menu_Select(Selected$)
   SET color 3
   SET cursor 1,1
   PRINT selected$
   IF selected$ = "Menu7" then EXIT DO
LOOP
CALL Erase_Menu

CALL Load_Menu_1(Menu_List$,letters$)
CALL Linear_Menu(Menu_List$,letters$)
DO
  CALL Linear_Menu_Select(Item_Selected$)
   SET color 3
   SET cursor 1,1
  PRINT Item_Selected$
  IF Item_Selected$ = "Menu12" then EXIT DO
LOOP
CALL Erase_Menu
END

EXTERNAL

SUB Load_Menu(array$(,),a$)
    DECLARE PUBLIC c_max              ! so I can used c_max
    DATA "Menu 1","Menu 2"
    DATA "Menu 3","Menu 4"
    DATA "Menu 5","Menu 6"
    DATA "Menu 7",""
    MAT array$ = nul$(4,2)
    MAT READ array$
     LET a$ = "ABCDEFG"               ! demo to change letter
selection
    LET MH$ = "- Testing Menu Selection Capability &
Changing Parameters -"
  CALL Change_Menu_SHARE(MH$,3,5,1,1,c_max,5)
END SUB                               ! end of 'Load_Menu'

SUB Load_Menu_1(array$(,),a$)
     DECLARE PUBLIC c_max             ! so I can used c_max
  DATA "Checkbook Deposit(s)","","Write a Check(s)","","
Void Transaction(s)",""
   DATA "Service Charge(s)","","Cash Deposit(s)","","
Cash Registration(s)",""
   DATA "Savings Deposit(s)","","Savings
Withdrawal(s)","","Account Balance(s)",""
  DATA "Hold Transaction(s)","","Change Checkbook","","
Quit Transactions",""
    MAT array$ = nul$(12,2)
    MAT READ array$
    LET MH$ = "- Linear Menu Selection Demo -"
    LET a$ = "DrVsCRswaHCQ"           ! example of mixed
upper/lower case
  CALL Change_Menu_SHARE(MH$,2,1,1,1,c_max,1)
END SUB                               ! end of 'Load_Menu_1'

SUB Buffer
    SET cursor "OFF"
     DO                               ! clear keyboard buffer
       IF key input then
          GET KEY b                   ! get the key input
          GET MOUSE x,y,z             ! get mouse input too
       ELSE
          GET MOUSE x,y,z             ! get mouse input
          IF z <> 3 and NOT key input then EXIT SUB
       END IF
    LOOP
END SUB                               ! end of 'Buffer'

MODULE Global
PUBLIC r_max,c_max
OPTION BASE 1
ASK max cursor r_max,c_max            ! max row and column of
output screen
END MODULE                            ! end of 'Global'


MODULE Menus                          ! collection of computer
independent menus
DECLARE PUBLIC r_max,c_max
!
SHARE col_1,col_2,col_3,col_4,Left_Len,Right_Len
SHARE Menu_Const,Menu_Odd,Menu_Space,Menu_Elements
SHARE Menu_Max_Row,Menu_Min_Row,Menu_Pen_Color,Menu_Size
SHARE Menu_Arrow_Color,Msg_Pen_Color,row_used,bar
SHARE Menu_Header$,Erase_Lines$,a$(1,2),menu$(1,4)
!
```

Workbench 2

Available Now!

Benchmark Modula-2 has long been recognized as the premier software development environment for the Amiga, combining the low-level machine access of C with the readability and ease of use of Turbo-Pascal. Now, by special arrangement with Aglet Software, the power of Workbench 2.04 is available to Benchmark Modula-2 programmers.

The package includes **AREXX**, **GadTools**: instant gadgets and menus, **BOOPSI**: object-oriented Intuition, **ASL**: standard requesters, **IEEE Math**, **IFFParse**: easy IFF access, and many **Example Programs**.

Benchmark WB2 offers the power of add-on libraries worth at least $500 for one special introductory price of only $125.

We're offering special discounts on upgrades to Benchmark for other Modula-2 users. Call or write for our free catalog.

Money-back guarantee on all Armadillo Computing products.

Armadillo Computing

5225 Marymount Drive, Austin, Texas 78723
Phone/Fax: 512/926-0360 BIX: jolinger
MasterCard and Visa accepted

```
LET Menu_Header$ = "Select by key or use mouse -"
LET Menu_Space = 1                  ! lines between menu
items
LET Menu_Pen_Color = 1              ! color used for print-
ing menu
LET Menu_Arrow_Color = 3            ! arrows
(Display_Menu_Select) or highlight color (Linear_Select)
LET Msg_Pen_Color = 4               ! color used for
Menu_Header$
LET gap = 5                         ! spaces between lines
menu items
LET Erase_Line$ = repeat$(" ",c_max)
IF len(Menu_Header$) > c_max then CAUSE EXCEPTION 104,
"Menu_Header$ character length is too long."

SUB Display_Menu(Menu_List$(,),letters$)
   LET Menu_Size = size(Menu_List$,1)
   LET Menu_Elements = len(letters$)    ! total # of
elements in array
   LET Menu_Const = int(Menu_Elements/2)    ! indicates
how many menu pairs there are
   LET Menu_Odd = remainder(Menu_Elements,2)    ! indi-
cates an odd row in array
   LET Menu_Min_Row = int((r_max - ((Menu_Const +
Menu_Odd) * Menu_Space - Menu_Space + 1 + 2))/2)
   LET letters$ = trim$(ucase$(letters$))
   IF Menu_Min_Row-2 < 1 then      ! menu will not fit on
screen
      CAUSE EXCEPTION 100, "Minimum row is less than
one."
   END IF
   LET Menu_Max_Row = Menu_Const * Menu_Space +
Menu_Min_Row - Menu_Space
   LET Left_Len, Right_Len = 0
   FOR n=1 to Menu_Size           ! find Max size of Left
and Right menu items
      IF len(Menu_List$(n,1)) > Left_Len then LET
Left_Len = len(Menu_List$(n,1))
      IF len(Menu_List$(n,2)) > Right_Len then LET
Right_Len = len(Menu_List$(n,2))
   NEXT n
   !
   ! calculate the print columns

   LET col_1 = int((c_max-(Left_Len + Right_Len + 11 +
4))/2)
   IF col_1 < 1 then CAUSE ERROR 101, "Column less than
one."
   LET col_2 = col_1 + Left_Len + 5 - 1
   LET col_3 = col_2 + col_1 + 1      ! allow for 1
blanks between left and right
```

```
   LET col_4 = col_3 + Right_Len + 5 - 1
   LET x = 0                        ! array counter
   LET j = 1                        ! letters$() counter
   FOR n=Menu_Min_Row to Menu_Max_Row step Menu_Space
      LET x = x + 1                 ! array counter
      SET color Menu_Pen_Color
      SET cursor n,col_1
      PRINT Menu_List$(x,1)         ! menu description, left
side
      SET cursor n,col_2
      PRINT letters$(j:j)           ! menu letter selection,
left side
      SET cursor n,col_3
      PRINT Menu_List$(x,2)         ! menu description,
right side
      SET cursor n,col_4
      PRINT letters$(j+1:j+1)       ! menu letter selection,
right side
      SET color Menu_Arrow_Color
      SET cursor n,col_2-1          ! set to print arrow
left side
      PRINT "->"
      SET cursor n,col_4-1          ! set to print arrow
right side
      PRINT "->"
      LET j = j + 2                 ! letters$() counter
   NEXT n
   IF Menu_Odd = 1 then             ! display one more menu
selection
      LET x = size(Menu_List$,1)      ! get max row
element of array
      SET color Menu_Pen_Color
      SET cursor Menu_Max_Row + Menu_Space,col_1
      PRINT Menu_List$(x,1)         ! display menu descrip-
tion
      SET cursor Menu_Max_Row + Menu_Space,col_2
      PRINT letters$(len(letters$):len(letters$))
      SET color Menu_Arrow_Color
      SET cursor Menu_Max_Row + Menu_Space,col_2-1   !
cursor for the arrow
      PRINT "->"
   END IF                           ! end of 'IF Menu_Odd =
1 ...'
   SET color Msg_Pen_Color
   SET cursor Menu_Min_Row - 2,int((c_max-
len(Menu_Header$))/2)    ! center point
   PRINT Menu_Header$
   MAT x$ = nul$(Menu_Size,2)
   LET k = 1                        ! initialize letters$()
counter
   FOR n=1 to Menu_Size
      LET x$(n,1) = using$("###",ord(letters$(k:k)))
      IF ord(letters$(k+1:k+1)) <> -1 then LET x$(n,2) =
using$("###",ord(letters$(k+1:k+1)))
      LET k = k + 2                 ! inc letters$() counter
   NEXT n
   LET row_used = Menu_Max_Row
   IF Menu_Odd = 1 then LET row_used = Menu_Max_Row +
Menu_Space
END SUB                             ! end of 'Display_Menu'

SUB Display_Menu_Select(Item_Selected$)
   !
   ! Display_Menu subroutine must be called first, then
this
   ! subroutine can be called as many times as you like
   !
   CALL Outline
   DO                                ! forever
      IF key input then
         GET KEY a                  ! get keyboard input
         IF a > 90 then LET a = a - 32     ! make upper
case
         LET position$ = using$("%%%",a)    ! put in same
format as x$ string
         FOR n=1 to Menu_Size
            IF position$ = x$(n,1) then
               LET Item_Selected$ = "Menu" &
using$("%%",n*2-1)   ! left menu selected
               EXIT SUB
            ELSEIF position$ = x$(n,2) then
               LET Item_Selected$ = "Menu" &
using$("%%",n*2)    ! right menu selected
               EXIT SUB
            END IF                  ! end of 'IF position$ =
x$ ...
         NEXT n
      ELSE                          ! no key input, see if
mouse was used
         GET MOUSE x,y,state
         IF state = 1 then
            LET row = y_max - round(y*(y_max-1))
            LET col = round(x*(x_max-1)) + 1
            FOR n=Menu_Min_Row to row_used step
Menu_Space
               IF row = n then
                  IF row >= Menu_Min_Row and row <=
row_used then
                     IF col >= col_1 and col <= col_2
then    ! left menu selected
                        IF Menu_Space > 1 then       !
special case for Menu_Space
                           LET Item_Selected$ = "Menu" &
using$("%%",(int((row-Menu_Min_Row)/(Menu_Space))+1)*2-1)
                        ELSE
                           LET Item_Selected$ = "Menu" &
using$("%%",(int((row-Menu_Min_Row)/(Menu_Space))+1)*2-1)
                        END IF
                        EXIT SUB
                     ELSEIF col >= col_3 and col <=
col_4 then
                        IF row <= row_used then
                           IF Menu_Space > 1 then     !
special case for Menu_Space
                              LET Item_Selected$ =
"Menu" & using$("%%",(int((row-Menu_Min_Row)/
Menu_Space)+1)*2-2)
                           ELSE
                              LET Item_Selected$ =
"Menu" & using$("%%",(int((row-Menu_Min_Row)/
Menu_Space)+1)*2)
                           END IF
                           EXIT SUB
                        ELSEIF row = row_used and
Menu_Odd <> 1 then    ! if is an odd row
                           IF Menu_Space > 1 then     !
special case for Menu_Space
                              LET Item_Selected$ =
"Menu" & using$("%%",(int((row-Menu_Min_Row)/
Menu_Space)+1)*2-1)
                           ELSE
                              LET Item_Selected$ =
"Menu" & using$("%%",(int((row-Menu_Min_Row)/
Menu_Space)+1)*2)
                           END IF    ! end of 'IF
Menu_Space > 1 ...
                           EXIT SUB
                        END IF
                     END IF     ! end of 'IF col >=
col_1 and col ...
                  END IF        ! end of 'IF row >=
Menu_Min_Row and ...
                  EXIT FOR      ! exit n for loop
               END IF           ! end of 'IF row = n ...
            NEXT n
         END IF                 ! end of 'if state = 1
      END IF                    ! end of 'IF key input
...
   LOOP
END SUB                                  ! end of
"Display_Menu_Select"

SUB Linear_Menu(array$(,),letters$)
   DIM t$(1)
   MAT t$ = nul$(c_max-3)       ! make it big enough to
hold a full screen minus Menu_Header$
   LET Menu_Size = size(array$,1)
   MAT menu$ = nul$(Menu_Size,3)     ! make working
menu$(,) same size as Menu_Size
   LET string$ = ""           ! initialize working
variable
   !
   ! load ASCII values for letters in each menu
   !
   FOR n=1 to Menu_Size
      LET menu$(n,1) = array$(n,1)   ! load in menu
array items
      LET menu$(n,2) = letters$[n:n]     ! load in
letter
   LET menu$(n,3) = using$("%%%",ord(letters$[n:n]))
! load in letter
      LET string$ = string$ & repeat$(" ",Gap) &
array$(n,1)  ! make a long string
   NEXT n
   !
   ! find the rows, and col1, and col2
   !
   LET k = 1                         ! initialize t$()
counter
   FOR n=1 to Menu_Size           ! take string$ apart and
load into t$()
      IF len(string$) < c_max then
         LET t$(k) = string$[1:len(string$)]
            LET max_t = k             ! remember how many
lines loaded into t$()
```

Memory Management, Inc.

Amiga Service Specialists

Over four years experience! Commodore authorized full service center. Low flat rate plus parts. Complete in-shop inventory.

Memory Management, Inc.
396 Washington Street
Wellesley, MA 02181
(617) 237 6846

```
          EXIT FOR
       END IF
     LET x = posr(t$(i_line),repeat$(" ",gap),ic_max)   !
work backwards to find gap blanks
       IF x = 0 or x = 1 then
          LET t$(k) = ltrim$(rtrim$(t_string$))
       ELSE                        ! x is past or 1
          LET t$(k) = t_string$[1:x-1]   ! load in a line of
menu selection
          LET t_string$ = ltrim$(t_string$[x:len(t_string$)])
       END IF
       LET k = k + 1               ! inc t$() counter
    NEXT n
  LET Menu_Min_Row = int((r_max - (k * Menu_Space + 2))/
2)
  IF Menu_Min_Row-2 < 1 then   ! menu will not fit on
screen
     CAUSE EXCEPTION 100, "Minimum row is less than
one"
  END IF
  LET Menu_Max_Row = Menu_Min_Row + k * Menu_Space -
Menu_Space
  LET start = gap+1             ! start at beginning, to
find the words
  !
  ! compute the row,col,pointer selection into
  !
```

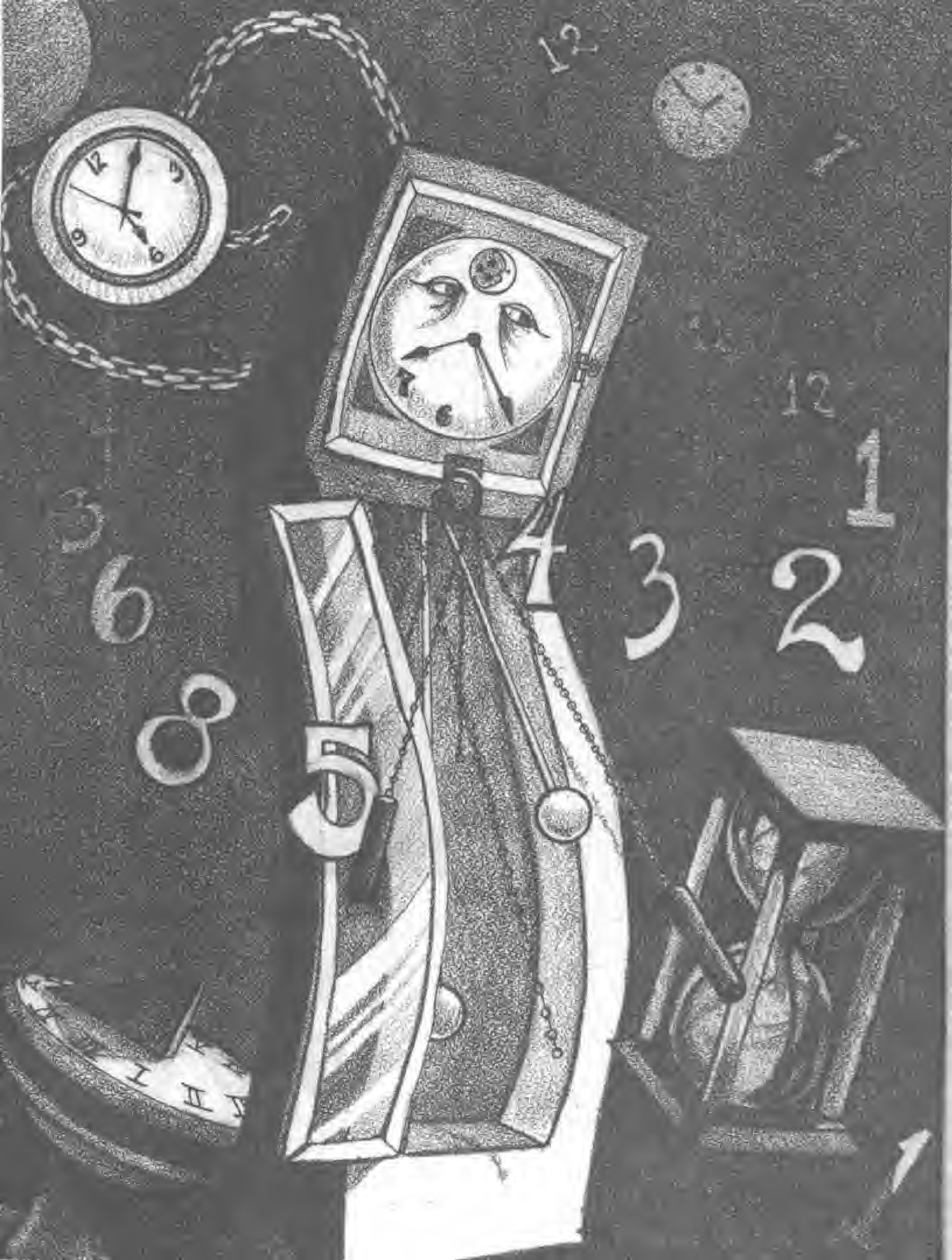
```
  LET row = Menu_Min_Row        ! starting screen row
  LET k, n = 1                  ! inc t$() counter
  DO                            ! forever
    LET x = pos(t$(k),menu$(n,2),start)  ! find start
of menu$(n,2)
      IF x <> 0 then
        LET start = x+len(menu$(n,2)) + 1   ! move start
up to next word
        LET menu$(n,4) = using$("##",row) &
using$("###",x) & using$("###",start)
         LET n = n + 1            ! inc array counter
         IF n > Menu_Size then EXIT DO
      ELSE                        ! x = zero
        LET row = row + Menu_Space   ! step down to next
menu row
         LET k = k + 1            ! inc the t$() counter
         IF k > max_t then EXIT DO
         LET start = gap+1        ! re-initialize
      END IF
    LOOP
    !
    ! display the menu
    !
    LET k = 1                     ! set t$() counter
   SET color Msg_Pen_Color
   SET cursor Menu_Min_Row-2, int((c_max-
len(Menu_Header$))/2)
   PRINT Menu_Header$
    LET begin = 1                 ! letter selection
counter
  FOR n=Menu_Min_Row to Menu_Max_Row step Menu_Space
      IF k > max_t then EXIT FOR
     SET color Menu_Pen_Color
      SET cursor n,1              ! on the row, n, and
column one
      PRINT t$(k)                 ! display line of menu
      LET start = gap+1           ! by-pass the initial
'gap' spaces at the beginning of string
      FOR j=begin to Menu_Size
         LET x = pos(t$(k),menu$(j,2),start)
           IF x = 0 then
             LET begin = j-1    ! set begin back one
since no selections taken on t$(k)
             LET k = k + 1      ! increment t$() counter
              EXIT FOR          ! exit the j for loop
           END IF
         SET color Menu_Arrow_Color     ! highlighted
letter color
          SET cursor n,x
          PRINT menu$(j,2)      ! print the highlighted
letter
        LET start = pos(t$(k),repeat$(" ",gap),x)
! move start past 'gap' blanks
          IF start = 0 then
            LET begin = j+1     ! set begin forward
since we used a letter
             LET k = k + 1      ! inc t$() counter
             EXIT FOR           ! exit j for loop
          END IF
      NEXT j
   NEXT n
END SUB                         ! end of 'Linear_Menu'

SUB Linear_Menu_Select(item_Selected$)
   CALL Buffer
  ! Must execute 'Linear_Menu' before using this subrou-
tine
```

```
    !
    ! get menu selection
    !
    DO                              ! forever loop
       IF key input then            ! check for key input
          GET KEY a
         LET A$ = using$("%%%",a)       ! string form of
ASCII key selected
          FOR n=1 to Menu_size      ! step through menu
ASCII options
              IF a$ = menu$(n,4) then
                 LET Item_Selected$ = "Menu" &
using$("%%",n)
                    EXIT SUB
                 END IF
            NEXT n
       ELSE                         ! no key input, check
for a mouse input
          GET MOUSE x,y,state
          IF state = 1 then         ! left mouse button
released
             LET row$ = using$("%%",(r_max-round(y *
(r_max-1))))   ! converts screen coordinates
             LET col$ = using$("%%",(1 + round(x * (c_max-
1))))      ! to row and column coordinates
              LET key$ = row$ & col$ & col$  ! key to
compare to menu$(k,4)
             FOR n=Menu_Min_Row to Menu_Max_Row step
Menu_Space
                 IF using$("%%",n) = row$ then    ! row
match
                    FOR k=1 to Menu_Size-1  ! determine if
col1 & col2
                        IF menu$(k,4) <= key$ and key$ <
menu$(k+1,4) then     ! key between values
                           IF menu$(k,4)[5:6] >= key$[5:6]
then ! key within menu$(k,4) range
                              LET Item_Selected$ = "Menu"
& using$("%%",k)
                                 EXIT SUB
                              END IF       ! end of 'IF
menu$(k,4)[5:6] ...'
                        END IF    ! end of 'IF menu$(k,4)
<= key$ ...'
                     NEXT k
                    IF key$ = menu$(Menu_Size,4) and
menu$(Menu_Size,4)[5:6] >= key$[5:6] then   ! selection
within range
                      LET Item_Selected$ = "Menu" &
using$("%%",Menu_Size)
                       EXIT SUB
                    END IF
                   EXIT SUB      ! exit n for structure
                 END IF          ! end of 'IF
using$("%%",n) ....'
              NEXT n
          END IF                    ! end of 'IF state
.........'
       END IF                       ! end of 'IF key input
.......'
    LOOP                            ! forever loop
END SUB                              ! end of
'Linear_Menu_Select'

SUB
Change_Menu_SHARE(MHS,MS,MPC,MAC,Msg_PC,Erase_Line_Len,gap_Ch)
    !
    ! used to change module SHARED variables
    !
   IF Erase_Line_Len > c_max then LET Erase_Line_Len =
c_max
    IF len(MHS) > c_max-1 then CAUSE EXCEPTION 101,
"Length of MHS to large."
    LET Menu_Header$ = MHS
    LET Menu_Space = MS
   LET Menu_Pen_Color = MPC
   LET Menu_Arrow_Color = MAC
   LET Msg_Pen_Color = Msg_PC
  LET Erase_Line$ = repeat$(" ",Erase_Line_Len)
   LET gap = gap_Ch
END SUB                              ! end of
'Change_Menu_SHARE'

SUB Erase_Menu
    !
   ! use only after calling 'Display_Menu' or
'Linear_Menu_Select'
    !
  SET cursor Menu_Min_Row-2,1
   PRINT Erase_Line$               ! erase Menu_Header$
  FOR n=Menu_Min_Row to row_used step Menu_Space
      SET cursor n,1
     PRINT Erase_Line$
   NEXT n
END SUB                              ! end of 'Erase_Menu'
END MODULE                           ! end of 'Menus'
```

✓

Complete source code and executables can be found on the AC's TECH disk.

Please write to:
T. Darrel Westbrook
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722



Time-Efficient Mathematical Algorithms For Producing Animation

by Robert Galka

Introduction

The AmigaDOS AnimOb structure contains several numeric fields used to produce animation quickly. These fields use what the *Amiga ROM Kernel Reference Manual* refers to as "Animations Special Numbering System." Two things make this numbering system special: the format of the numbers and the operations performed on them. This paper discusses and generalizes this numbering system. After you have read it, you should be able to:

1. Understand how this numbering system is related to the continuous linear equations and continuous quadratic equations taught in most algebra courses.
2. Choose numbers and operations to cause desired animation effects with or without using the AnimOb structure.
3. Extend these operations to periodic functions.
4. Construct tables for periodic functions, and use these tables to produce animation.

This paper is not written for mathematicians. The mathematics is developed intuitively. Well-known identities are given without proof. An Amiga programmer with an understanding of algebra should be able to apply these techniques to her programs.

The paper covers:
1. A review of fixed-point numbers and some operations performed on them.
2. A discussion of the equations used to change an attribute of an object at a constant rate. These are linear equations.
3. A discussion of the equations used to change an attribute of an object at a rate of change that changes at a constant rate. These are quadratic equations.
4. A review of certain trigonometric functions and identities. A method is developed for rotating objects by building a table and retrieving values from the table.
5) A discussion of periodic functions and the domains of these functions.
6) A discussion of the software for this article which is included on *AC's TECH* Disk.

Fixed-Point Numbers

Fixed-point binary numbers are of the form In...I0pF1...Fm where the Is make up the integer part, p is the radix (binary) point, and the Fs form the fractional part of the number. There is a fixed, predetermined number of digits on either side of the radix point. The programmer knows where the radix point lies. If there are no F's in the number, then the fixed-point binary number is an integer.

Although any number of bits and representation (2s complement, sign magnitude, 1s complement) may be used to represent fixed-point binary numbers on the Amiga computer, they are represented most naturally as signed characters, short integers, integers, or long integers.

Fixed-point numbers may be used in applications that don't require a "floating" radix point and don't require very large or very small numbers. Software which produces animation is an example of such an application. If fact, the velocity fields and acceleration fields in the Amiga's AnimOb structure are defined as WORD's, but they are really fixed-point binary number with six digits to the right of the radix point.

►

Table 1 below shows the relationship between real numbers and a certain fixed-point representation of them. Notice that fixed-point numbers may introduce round-off errors, overflow and underflow. It is important to pick the size of the fixed-point number representation and the number of fractional bits so that these problems don't occur in your application. In Table 1, the fixed-point numbers are 8-bits long with 3 bits to the right of the radix point.

TABLE 1

real (base 10) number	fixed-point number	comment
1.0	00001 000	
1.5	00001 100	
-1.0	11111 000	(2s complement)
1/125	00000 001	
1.1	00001 001	(rounded)
4.5	00100 100	
100.0	********	(overflow)
-100.0	********	(overflow)
0.0001	00000 000	(underflow)

Some facts about operations on fixed-point numbers are:
1. Two fixed-point numbers may be added using integer addition provided they have the same number of digits to the right of the radix point.
2. Two fixed-point numbers may be subtracted using integer subtraction provided they have the same number of digits to the right of the radix point.
3. If A is a fixed-point number with n digits to the right of the radix point, and B is a fixed-point number with m digits to the right of the radix point, then C = A*B is a fixed-point number with n+m digits to the right of the radix point.

The programs contained on the distribution diskette use fixed-point numbers and the algorithms developed below to produce animation quickly on the Amiga computer.

Linear Equations

If an attribute of an object changes at a constant rate, then that attribute may be described by the function L defined by the equation

```
L(t) = (b*t) + c    [1]
```

where b is the rate of change and c is L(0). The independent variable, t, represents time. Time may be expressed in any convenient units such as seconds, years, or video frames. L(t) is dependent on t and represents the attribute. L(t) may be measured in units such as feet, radians, pixels, or kilohertz.

If the points {(t, L(t)) | t is a real number} are plotted, the result is a straight line. The values of the constants, b and c, may be determined from two points on the line, or by one point on the line and the rate of change, or slope, of the line.

If two points, (t0, L(t0)) and (tn, L(tn)) are known, then

```
b = (L(tn)-L(t0)) / (tn-t0)    [2]
c = L(t0) - (b*t0)
```

If one point, (t0, L(t0)), and the rate of change, r, are known, then

```
b = r    [3]
c = L(t0) - (b*t0)
```

Equation [1] is perfectly general in the sense that for any value of t, the value L(t) may be computed directly. For example, if L(t) = 10*t+7, then L(1.5) = 10*1.5+7 = 22. It is not necessary to keep a history of previously-computed values of L(t) to compute future values of L(t). Equation [1] requires one addition and one multiplication to compute L(t) for any t. Multiplication is a time-intensive operation on the Amiga computer. Without considering the various addressing modes of the 68000 processor, the *Motorola M68000 Programmer's Reference Manual* shows that a long word ADD instruction takes about six clock cycles, while the MULS instruction takes about 70 clock cycles.

If t is restricted to values of the form t0 plus non-negative integers (t0, t0+1, t0+2, t0+3, ...), and L(t) is computed before L(t+1) is computed, another function, RL (Recursive Linear), and a constant, DeltaRL, may be used to compute L(t) over this restricted domain. RL is more efficient than L in the sense it uses a single addition and a previously computed value to compute RL(t+1).

RL and DeltaRL are defined in terms of the coefficients of L(t) = b*t+c and t0 as follows:

```
  DeltaRL = b;          [4]
  RL(0) = b*t0 + c = L(t0);
  if t is a non-negative integer then
               RL(t+1) = DeltaRL+RL(t)

Note that RL(t+1) = DeltaRL + RL(t)              [5]
                  = DeltaRL*2 + RL(t-1)
                  = DeltaRL*3 + RL(t-2)
                  = ...
                  = DeltaRL*(t+1) + RL(0)
                  = b*(t+1) + b*t0 + c
                  = b(t0+t+1) + c
                  = L(t0+t+1)
```

So, RL and DeltaRL may be used to compute L(t) for t = t0, t0+1, t0+2, t0+3,...

Quadratic Equations

If the rate of change of an attribute of an object changes at a constant rate, then that attribute may be described by the function Q defined by the quadratic equation

```
Q(t) = a*t*t + b*t + c          [6]
     = (a*t + b)*t + c
```

where a is not zero and is proportional to the rate of change of the rate of change of the attribute, b is the rate of change of the attribute at t=0, and c is the value of Q(0). The independent variable, t, represents time. Q(t) is dependent on t and represents the attribute.

When the points {(t, Q(t)) | t is a real number} are plotted, the result is a parabola. The values of the constants a, b, and c may be determined from three points on the parabola or by two points on the parabola and the slope, or first derivative, at any point on the parabola.

If three points, (t0, Q(t0)), (tm, Q(tm)), and (tn, Q(tn)) are known, then

```
a = ( (Q(tn)-Q(tm)) / (tn-tm) -              [7]
      ((Q(tm)-Q(t0)) / (tm-t0)) ) / (tn-t0)

b = ((Q(tm)-Q(t0)) / (tm-t0)) - (a * (tm+t0))

c = Q(t0) - (a*t0 + b)*t0
```

If two points, (t0, Q(t0)) and (tn, Q(tn)), and the slope, Vm, of any point, (tm, Q(tm)), are known, then

```
a = ((Q(tn)-Q(t0)) - (Vm*(tn-t0))) / ((tn+t0-2*tm)*(tn-t0))        [6]

b = Vm - 2*a*tm

c = Q(t0) - (a*t0 + b) * t0
```

Equations [6] are perfectly general in the sense that for any value of t the value Q(t) may be computed directly. It is not necessary to store a history of previously computed values of Q(t) to compute future values of Q(t). The last expression in [6] requires two additions and two multiplication to compute Q(t) for any value of t.

If t is restricted to values of the form t0 plus non-negative integers (t0, t0+1, t0+2, t0+3,...), and two values related to Q(t) are compute before Q(t+1) is computed, two other functions, RQ (Recursive Quadratic) and DeltaRQ, and one constant, Delta2RQ, may be used to compute Q(t) over the restricted domain. These are more efficient than Q in the sense that they use two additions and previously computed values, RQ(t) and DeltaRQ(t), to compute RQ(t+1).

RQ, DeltaRQ, and, Delta2RQ, are defined is terms of the coefficients of Q(t) = a*t*t + b*t + c and t0 as follows

```
Delta2RQ = 2*a          [9]

DeltaRQ(0) = a*(2*t0-1) + b
```
if t is a non-negative integer then
```
    DeltaRQ(t+1) = DeltaRQ(t) + Delta2RQ

RQ(0) = a*t0*t0 + b*t0 + c = Q(t0)
```
if t is a non-negative integer then
```
    RQ(t+1) = RQ(t)+DeltaRQ(t+1)
```

To convince the reader that Q(t) is being computed for t = t0, t0+1, t0+2,... we will compute a few values of RQ(t) and see a pattern emerge. Using the pattern, we'll write a closed expression for RQ(t) and show that it is equal to Q(t0+t). The mathematically sophisticated reader my want to use mathematical induction to prove this to herself.

It should be clear from the definition of DeltaRQ that

```
DeltaRQ(t) = DeltaRQ(0) + t*Delta2RQ
```
for all non-negative integers t

Using this fact we have

```
RQ(0) = a*t0*t0 + b*t0 + c
RQ(1) = RQ(0) + DeltaRQ(1)
      = RQ(0) + DeltaRQ(0) + Delta2RQ
RQ(2) = RQ(1) + DeltaRQ(2)
      = RQ(0) + DeltaRQ(0) + Delta2RQ + DeltaRQ(0) + 2*Delta2RQ
      = RQ(0) + 2*DeltaRQ(0) + 3*Delta2RQ
RQ(3) = RQ(2) + DeltaRQ(3)
      = RQ(0) + 2*DeltaRQ(0) + 3*Delta2RQ + DeltaRQ(0) + 3*Delta2RQ
      = RQ(0) + 3*DeltaRQ(0) + 6*Delta2RQ
RQ(4) = RQ(0) + 4*DeltaRQ(0) + 10*Delta2RQ
```

The pattern we see is

```
RQ(t) = RQ(0) + t*DeltaRQ(0) + ((t*(t+1))/2)*Delta2RQ
      = (a*t0*t0 + b*t0 + c) + (a*(2*t0-1)+b)*t + 2*a*((t*(t+1))/2)
      = (a*t0*t0 + b*t0 + c) + (2*a*t0*t -a*t + b*t) + (a*t*t + a*t)
      = a*(t0*t0 + 2*t0*t + t*t + t - t) + b*(t0+t) + c
      = a*(t0+t)*(t0+t) + b*(t0+t) + c
      = Q(t0+t)
```

So, RQ, DeltaRQ and Delta2RQ may be used to compute Q(t) for t = t0, t0+1, t0+2, ...

Quick Sine and Cosine Functions

The sine function and the cosine function are used to rotate objects. If (x, y) is a point in a Cartesian coordinate system, the formulae to rotate it r radians about the origin are

```
newx = x*cos(r) - y*sin(r)          [11]
newy = x*sin(r) + y*cos(r)
```

If r is positive, the point is rotated counterclockwise. Otherwise it is rotated clockwise

The usual software implementation of these functions is expensive in the sense that it requires one of more multiplications and additions to compute the sine or cosine of any angle. In this section, two new functions Qsin (quick sine) and Qcos (quick cosine) are defined. These new function are related to the sine and cosine, but their domain is the set of integers and their period is an integral power of two. This makes it easy to construct a table of values and look up the values of Qsin(t) and Qcos(t) in that table.

Below are some important trigonometric identities. These identities will be used later to find the values of Qsin and Qcos in a table of length one fourth the period of Qsin

```
cos(r) = sin(PI/2 - r)          [12]
sin(r) = sin(PI - r)
cos(r) = -sin(r - PI/2)
sin(r) = -sin(r - PI)
cos(r) = -sin(3*PI/2 - r)
sin(r) = -sin(2*PI - r)
cos(r) =  sin(r - 3*PI/2)
```

Now, define a constant FPPI (Fixed Point PI) and the two functions Qsin and Qcos as follows:

FPPI = 2**N for some positive integer N. The value of N depends on the resolution desired

For integer values t, define Qsin(t) and Qcos(t) as follows:

```
Qsin(t) = sin(t * ((2*PI)/(2*FPPI)))          [13]
Qcos(t) = cos(t * ((2*PI)/(2*FPPI)))
```

Qsin and Qcos are like the sine and cosine functions with their domain restricted to increments of (2*PI) / (2*FPPI) radians. The period of Qsin and Qcos is 2*FPPI, and the domain is the set of integers.

The following identifies hold because the equivalent identities [12] hold for the trigonometric functions used to define Qsin and Qcos

```
Qcos(t) = Qsin(FPPI/2 - t)          [14]
Qsin(t) = Qsin(FPPI - t)
Qcos(t) = -Qsin(t - FPPI/2)
Qsin(t) = -Qsin(t - FPPI)
Qcos(t) = -Qsin(3*FPPI/2 - t)
Qsin(t) = -Qsin(2*FPPI - t)
Qcos(t) =  Qsin(t - 3*FPPI/2)
```

Identities [14] are used in algorithm [15] below to find the value of Qsin(t) or Qcos(t) in a table of size (FPPI/2) +1

Create a table, QsinTable, of length (FPPI/2)+1, and store Qsin(t) in QsinTable[t]. Depending on your application, the values stored in the table may be floating point values or fixed point values.

Algorithm [15] below finds Qsin(t) and Qcos(t) in QsinTable for values of t equal to or greater than 0 and less than 2*PPPI. In the next section, it is shown that this restriction on the values of t is a reasonable restriction.

Qsin and Qcos Table Lookup Algorithm

```
if ( t <= PPPI/2 )        (15)
{
  Qsin(t) = QsinTable(t)
  Qcos(t) = QsinTable(PPPI/2 - t)
}
else
{
  if ( t <= PPPI )
  {
    Qsin(t) = QsinTable(PPPI - t)
    Qcos(t) = -QsinTable(t - PPPI/2)
  }
  else
  {
    if ( t <= 3*PPPI/2 )
    {
      Qsin(t) = -QsinTable(t - PPPI)
      Qcos(t) = -QsinTable(3*PPPI/2 - t)
    }
    else
    {
      Qsin(t) = -QsinTable(2*PPPI - t)
      Qcos(t) = -QsinTable(t - 3*PPPI/2)
    }
  }
}
```

Periodic Functions

A non-constant function, Per, is a periodic function if there exists a constant c in the domain of Per such that Per(t) = Per(t+c) for all t in the domain of Per. The smallest such number, p, is called the period of Per. It follows from the definition of periodic function that Per(t) = Per(t+i*p) for every integer i. Examples of periodic functions are sine, cosine, sawtooth, triangle and square wave.

An attribute of an object may be changed by applying a periodic function to that attribute and modifying the input of the periodic function. In this section, functions RLP (Recursive Linear Periodic) and RQP (Recursive Quadratic Periodic), which vary the input of periodic functions, are derived. RLP is derived from RL. RQP is derived from RQ. The output (range) of these new functions is always equal to or greater than zero and less than the period of the periodic function. RLP is derived first.

Let Per be a periodic function with period p. Suppose that the domain (input) of Per is the range (output) of the function RL shown in [4]. Define a new function RLP and a constant DeltaRLP as follows:

```
DeltaRLP = DeltaRL mod p          (16)

RLP(0) = RL(0) mod p
```

if t is a non-negative integer and RLP(t)+DeltaRLP < p then

```
RLP(t+1) = RLP(t)+DeltaRLP
```

if t is a non-negative integer and RLP(t)+DeltaRLP >= p then

```
RLP(t+1) = RLP(t)+DeltaRLP - p
```

It should be clear from the definition of RLP that 0 <= RLP(t) < p for all non-negative integers t. It also follows from [16] that for each non-negative integer t, there is an integer m such that RL(t) = RLP(t) + m*p. So,

```
Per(RL(t)) = Per(RLP(t) + m*p) = Per(RLP(t))
```

Equations [16] is a time-efficient algorithm which varies the input of a periodic function linearly and restricts the value of the input between zero and the period of the function. If the domain of the periodic function is a subset of integers or fixed-point numbers, then it is easy to use a table to determine the value of the function Per(t) for any t.

If the domain of Per is the range of the function RQ shown in [9], then define two new functions RQP and DeltaRQP, and a constant Delta2RQP as follows:

```
Delta2RQP = Delta2RQ mod p        (17)

DeltaRQP(0) = DeltaRQ(0) mod p
```

If t is a non-negative integer and DeltaRQP(t)+Delta2RQP < p then

```
DeltaRQP(t+1) = DeltaRQP(t)+Delta2RQP
```

If t is a non-negative integer and DeltaRQP(t)+Delta2RQP >= p then

```
DeltaRQP(t+1) = DeltaRQP(t)+Delta2RQP - p

RQP(0) = RQ(0) mod p
```

If t is a non-negative integer and RQP(t)+DeltaRQP(t+1) < p then

```
RQP(t+1) = RQP(t)+DeltaRQP(t+1)
```

If t is a non-negative integer and RQP(t)+DeltaRQP(t+1) >= p then

```
RQP(t+1) = RQP(t)+DeltaRQP(t+1) - p
```

Equations [17] is a time-efficient algorithm that varies the input of a periodic function quadratically and restricts the value of the input between zero and the period of the function. If the domain of the periodic function is a subset of integers or fixed-point numbers, then it is easy to use a table to determine the value of the function Per(t) for any t.

Software on the Distribution Diskette

It is time to apply this theory by looking at a few examples. The distribution diskette contains one command file: make-examples, three linker option files: example1.lnk, example2.lnk, and example3.lnk, three executable files: example1, example2, and example3, and six source files: display.c, my_protos.h, tools.c, example1.c, example2.c, and example3.c.

Make-examples is a command file that compiles and links the three programs on the diskette. Version 6.2 of the SAS/C Development System is used to create the programs. Use the AmigaDOS execute command to run make-examples from the CLI. The three linker option files are used by make-examples to link the object files into three executable files. The three executable files are described in detail below, after the descriptions of the source files.

Display.c contains all the functions necessary to initialize and cleanup a double-buffered lowres screen. The screen is 640 pixels wide by 200 pixels long. Information about double-buffered screens may be found in the *Amiga ROM Kernel Reference Manual: Libraries*.

My_Protos.h contains the function prototypes for the global functions in display.c and tools.c. My_protos.h is included in example1.c, example2.c, and example3.c.

Tools.c contains a fixed-point math toolkit. The functions in this file are:

FPToLongInt - converts a fixed-point number to an integer by rounding the fixed-point number to an integer.
LongIntToFP - converts an integer to a fixed-point number.
DoubleToFP - converts a double to a fixed-point number.
RLFm2Pts - computes RL(0), DeltaRL, and the coefficients b and c from two points on a line. RL(0) and DeltaRL are fixed-point numbers. b and c are double floating point numbers.
RQFm2PtsAndVel - computes RQ(0), DeltaRQ(0), Delta2RQ, and the coefficients a, b, and c from two points and the slope of a point on a parabola.
RQ(0), DeltaRQ(0), and Delta2RQ are fixed-point numbers. a, b and c are double floating point numbers.
RLPFm2Pts - computes RLP(0), DeltaRLP, and the coefficients b and c for two points on a line. The input to this function is NOT the two points. It is an initial angle (t0, s0), a final angle (tn, sn), and the number of rotations, nrot, to use to move from the initial angle to the final angle. The two points on the line are (t0, s0) and (tn, sn+2*PI*nrot). s0 and sn are double floating point numbers whose units are radians. RLP(0) and DeltaRLP are fixed-point numbers. b and c are double floating point numbers.
InitQsinTable - initializes a QsinTable of 1025 points.
LookupQsinQcos - finds a Qsin value and a Qcos value in the QsinTable by using the algorithm in the article.
InitAspectRatio - computes the aspect ratio as a fixed-point number. The aspect ratio is used to convert from world coordinates to screen coordinates.
QRotate - rotates a set of points, given in world coordinates, around the origin, converts the rotated set of points to screen coordinates, and translates the points in the X and Y directions by an amount specified by the caller of QRotate.
ComputeNextRL - computes RL(t) for t greater than 0.
ComputeNextRQ - computes RQ(t) for t greater than 0.
ComputeNextRLP - computes RLP(t) for t greater than 0.

Example1.c, example2.c, and example3.c contain example programs. The first example applies different types of motion to an attribute of similar objects. The second example applies different types of motion to two attributes of an object. The third example uses Qsin and Qcos to rotate an object. The examples are covered in detail below.

Example 1 - Applying Different Types of Motion to an Attribute of Similar Objects

The file example1.c contains the code for the first example. In this example, four sets of vertical lines are drawn across the screen horizontally. The lines are drawn from left to right. For each of 100 consecutive frames, one line from each set of lines is drawn. A different function is used to compute the positions of the lines in each set.

Run the first example and watch the lines being drawn on the screen. In the top row of lines, adjacent lines are equally spaced because a linear function, RL, is used to compute the position of each line. Linear functions cover equal distances in equal times.

Now look at the second set of lines. The lines are spaced closer together on the left side of the screen and farther apart on the right. Since speed is distance covered divided by time, we could say that the lines are moving faster as they are drawn from left to right. The slope, or first derivative, of an equation is the velocity of an object whose movement is described by the equation. The speed is the absolute value of the velocity. The quadratic equation, Q(t), used to determine the position of each line was chosen so that the slope is 0 for t=0. The absolute value of the slope increases as t increases. If you run the program again and compare the drawing of the first two sets of lines, it will appear that, initially, the top set of lines is being drawn faster than the second set, but the drawing of the second set accelerates and catches up to the first set when the last two lines are drawn.

The lines in the third set are spaced closer together on the right side of the screen and farther apart on the left. The quadratic equation, Q(t), used to determine the position of each line was chosen so that the slope is 0 for t=100. The absolute value of the slope decreases as t goes from 0 to 100.

Finally, the lines in the last set are spaced closer together on the left and right sides of the screen, and farther apart in the middle. Two quadratic equations, Q1(t) and Q2(t), are used to determine the positions of the lines in this set. The first quadratic equation is used to compute the position of each line on the left half of the screen. It has a slope of 0 at t=0. The absolute value of the slope increases as t increases. The second quadratic equation is used to compute the position of each line on the right half of the screen. Its slope is 0 at t=100. The absolute value of the slope of Q2(t) decreases as the value of t increases to t=100. This fourth set of lines appears to be drawn slowly at first, faster in the middle, and slowly again on the right side end of the screen.

Now examine the file example1.c. Near the top of it, a structure qmath_type is defined. x, dx, and d2x are fields containing RL(t) and DeltaRL, or RQ(t), DeltaRQ(t), and Delta2RQ. These values are stored as fixed-point numbers. ints contains the rounded integer portion of the value in x. ints is used to draw the lines on the screen because the positions of pixels are given as whole numbers. The coefficients for the continuous equations, x = b*t+c and x = a*t*t+b*t+c are stored in a, b, and c. These values are not used in this program but are included in case the reader wants to modify the program to examine these values. For real-world applications, such as games, the structure qmath should contain additional fields for information such as start and end times, parameter-attribute identifiers, and motion-type identifiers. The structures should probably be ordered by start time in one or more linked lists.

Now look at the function main in example1.c. main calls functions to initialize the display, to create the initial values for the functions used to produce animation, to produce animation, and to cleanup the display.

CreateFunctions calls functions contained in tools.c to initialize the qmath_type structures. RLFm2Pts is called to produce the initial values used to draw the top set of lines. The input to this function is two points. The output is RL(0), DeltaRL, the integer portion of RL(0), and the coefficients of a linear equation containing the points. CreateFunctions also makes four calls to RQFm2PtsAndVel to produce the initial values used to draw the remaining three sets of lines. There are four calls rather than three because the last set of lines is drawn using two equations instead of one.

MakeMotion draws the lines on the screen. Four lines, one from each set, is drawn in each frame. Look at the for loop in MakeMotion. Three similar lines of code are repeated four times. First, Move and Draw are called to draw a line on the screen. Next, ComputeNextRL or ComputeNextRQ is called to compute the position of the next line in

MOVING?

SUBSCRIPTION PROBLEMS?

Please don't forget to let us know. If you are having a problem with your subscription or if you are planning to move, please write to:

Amazing Computing Subscription Questions
PiM Publications, Inc.
P.O. Box 2140
Fall River, MA 02722

Please remember, we cannot mail your magazine if we do not know where you are.

Please allow four to six weeks for processing.

the set of lines. ComputeNextRL is called if the position of each line in the set is determined by a linear equation. ComputeNextRQ is called if the position of each line in the set is determined by a quadratic equation. The conditional statement within the for loop is used to draw the bottom set of lines on the screen. The 'if' part is used to draw the lines on the left side of the screen; the 'then' part is used to draw the lines on the right side of the screen.

Example 2 - Applying Different Types of Motion to Two Attributes of One Object

The file example2.c contains the code for the second example. A rectangle is animated for 121 frames. It moves both horizontally and vertically on the screen. A linear equation is used to compute the horizontal positions of the rectangle. A quadratic equation is used to compute the vertical positions of the rectangle. The quadratic equation was chosen to have a slope of zero when the rectangle reaches its highest point on the screen.

Examine the file example2.c. RQFm2PtsAndVel is called with two points and a slope of zero at the second point. The first point specifies the initial vertical position of the rectangle. The second point specifies the highest vertical position of the rectangle. It is the highest position because the slope at that point is zero and the first point is below it.

When you run this example, you'll see that the rectangle climbs from its lowest vertical position to its highest vertical position with decreasing speed, and it descends back to the bottom of the screen with increasing speed. Simultaneously, it is moving across the screen at a constant speed. This is the same type of motion you see when an object is thrown in the air.

The structure of the code for this example is similar to the structure of the code for the first example. The major differences are:

1. In the first example Move and Draw are used to draw lines. In this example RectFill is used to draw rectangles.
2. In the first example only one bit map is used. In the second example both bit maps are used.
3. In the first example the screen is not erased before the next set of lines is drawn. In this example the screen is erased before the next rectangle is drawn.

Example 3 - Rotating an Object

The file example3.c contains the code for the third example. An hour-glass figure is rotated 5 1/4 turns counterclockwise in 301 frames. The figure is specified in world coordinates. The only real requirement of our world coordinate system is that both axes use the same scale. The amount that the angle of rotation changes each frame is determined by a linear equation. A linear periodic function, RLP, is used to change the value of the angle. Since the algorithm for RLP is used, the value of the angle is never greater than the period of Qsin. Qsin and Qcos are used to rotate the figure about the origin. After the figure is rotated, it is converted to screen coordinates and translated to the center of the screen. For our purposes, the translation is necessary because the origin of the screen coordinate system is on the upper left corner of the display. Note that instead of translating the figure by a constant amount each frame, a linear function, RL, or a quadratic function, RQ, could have been used to move the figure on the screen.

Look at the functions in example3.c. In addition to initializing the display, creating RLP(0) and DeltaRLP, and updating the values once per frame, main calls functions to initialize the aspect ratio in fixed-point format and to initialize the QsinTable. Once per frame, the function MakeMotion calls SetRast to clear the display; calls QRotate to rotate and translate the figure, and to convert the figure to screen coordinates; calls Move and PolyDraw to draw the figure on the screen; and calls ComputeNextRLP to compute the angle of rotation for the next frame. Double-buffered bitmaps are used in this example.

Experiments to Perform

The reader may want to try the experiments described below.

1. Decrease the value of LIN_FRAC_BITS, the number of bits to the right of the radix point, and run the programs again. If LIN_FRAC_BITS becomes too small, the figures will not move correctly because of round-off error.
2. Decrease the value of FPPI and related constants and run example three again. When these values become too small, the image will not rotate smoothly.
3. Write the functions RQPFm2PtsAndVel and ComputeNextRQP. RQPFm2PtsAndVel computes RQP(0), DeltaRQP(0), and Delta2RQP. ComputeNextRQP computes RQP(t) for t greater than zero. Modify the third example so that the object starts rotating slowly, accelerates until the object rotates 2 and 1/8 times, and decelerates until it reaches a speed on zero at 5 and 1/4 rotations.

☑

display.c

```
/* display.c */
/* Copyright [illegible] by Robert Gelka */

/*****************************************************************
/* display.c contains all of the functions necessary to
   initialize and cleanup a double buffered Intuition
   screen. Information on double buffered screens can
   be found in the AMIGA ROM Kernel Reference Manual:
   Libraries.
 */
*****************************************************************/

#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>

#include <clib/exec_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>

#define SCREEN_WIDTH    (640)
#define SCREEN_HEIGHT   (200)
#define SCREEN_DEPTH    (2)

struct BitMap *bit_map[2];
struct Screen *screen;

static void CleanupBitMap (struct BitMap *bm)
{
  short idx;

  if (bm != NULL)
   {
    for (idx = 0; idx < SCREEN_DEPTH; idx++)
     {
      if (bm->Planes[idx] != NULL)
       {
        FreeRaster (bm->Planes[idx],
                    SCREEN_WIDTH, SCREEN_HEIGHT);
       }
     }
 FreeMem (bm, sizeof(struct BitMap));
  }
}

void CleanupDisplay (void)
{
  if (screen != NULL)
   {
    CloseScreen(screen);
   }
  CleanupBitMap (bit_map[0]);
  CleanupBitMap (bit_map[1]);
}

static int SetupBitMap(struct BitMap **bm)
{
  int status;
 short idx;

 *bm = (struct BitMap *)
      AllocMem (sizeof(struct BitMap), MEMF_CLEAR);

 if (*bm != NULL)
  {
   status = TRUE;
  InitBitMap (*bm, SCREEN_DEPTH,
             SCREEN_WIDTH, SCREEN_HEIGHT);
   for (idx = 0; idx < SCREEN_DEPTH; idx++)
    {
     (*bm)->Planes[idx] =
          (PLANEPTR)AllocRaster (SCREEN_WIDTH,
                                  SCREEN_HEIGHT);
     if ((*bm)->Planes[idx] != NULL )
      {
      BltClear((*bm)->Planes[idx],
               (SCREEN_WIDTH / 8) * SCREEN_HEIGHT, 1);
      }
      else
      {
       status = FALSE;
       break;
      }
    }
  }
 else
  {
   status = FALSE;
  }
 return(status);
}

int InitDisplay (void)
{
 int status;
 struct NewScreen new_screen;

 status = SetupBitMap (&bit_map[0]);
 if (status == TRUE)
  {
   status = SetupBitMap (&bit_map[1]);
   if (status == TRUE)
    {
     new_screen.LeftEdge=0;
     new_screen.TopEdge=0;

    new_screen.Width=SCREEN_WIDTH;
    new_screen.Height=SCREEN_HEIGHT;
    new_screen.Depth=SCREEN_DEPTH;

     new_screen.DetailPen=0;
     new_screen.BlockPen=1;

    new_screen.ViewModes=HIRES;
    new_screen.Type=CUSTOMSCREEN |
                     CUSTOMBITMAP | SCREENQUIET;
     new_screen.Font=NULL;
    new_screen.DefaultTitle=NULL;
    new_screen.Gadgets=NULL;
    new_screen.CustomBitMap=bit_map[0];

     screen = OpenScreen(&new_screen);
      if (screen != NULL)
       {
        screen->RastPort.Flags = DBUFFER;
        }
      else
        {
         Status = FALSE;
        }
      }
  }
 return (Status);
}
```

my_protos.h

```
/* my_protos.h */
/* Copyright 1992 by Robert Galka */

/
*************************************************************)
/* Prototypes for functions from tools.c */
/
*************************************************************)

long RtPhiVis (double fy, double x0,
              double fn, double xp,
              double *b, double *c,
              long *RtP, long *DeltaRtP);

long RtPhiFromRdVn (double x0, double y0,
                    double fn, double x2,
                    double fm, double ym,
                    double *a, double *b, double *c,
                    long *RQ, long *DeltaRQ,
                    long *Delta2RQ);

long RtRadius (double r1, double x0,
              double fn, double xo
              double rad,
              double *b, double *c,
              long *RtR, long *DeltaRtR);

void InitQuadTable (void);

void Quadrant (long Angle, long Tx, long Ty,
           short *Left, short *Right, long RPixel);

long ComputeNextRt (long *Rt, long DeltaRt);

long ComputeNextRQ (long *RQ, long *DeltaRQ,
                              long Delta2RQ);

long ComputeNextRtR (long *RtR, long DeltaRtR);

void InitAspectRatio (double Xdim, double Ydim);


/
*************************************************************)
/* Prototypes for functions from display.c */
/
*************************************************************

int InitDisplay (void);

void CleanupDisplay (void);
```

tools.c

THIS LISTING IS NOT COMPLETE AS SHOWN

```
/* tools.c */
/* Copyright 1992 by Robert Galka */

/
*************************************************************
/* tools.c contains all of the fixed point math functions.
 The static real time functions that compile inline
code
  use the __inline keyword to generate inline code. *
/
*************************************************************

#include <math.h>

/
*************************************************************
/* Important Constants */

/* Number of bits to right of radix point
          for most fixed point numbers */
#define LIN_FRAC_BITS   (12)
/* Number of bits to right of radix point
          for Quad. Gen. */
#define RQ_FRAC_BITS   (18)

/* Constants used to Gain and Quadrant operations */
/* Think of FPPI as Fixed Point representation of PI */
#define FPPI_D2         (1024) /* FPPI/2 */
#define FPPI           (2048)
#define FPPI_3D2        (3072) /* FPPI*(3/2) */
#define FPPI_T2         (4096) /* FPPI*2 */
#define FPPI_T2_FRAC    (FPPI_T2 << LIN_FRAC_BITS)

/* Note that PI (3.14...) is defined in math.h */
/
*************************************************************


static long QuadTable[FPPI_D2 + 1];           /* fixed point
format */
static long AspectRatio;                          /* fixed
point format */



static long __inline FPRoundInt (long FPVal,
                              long NbrFracBits)
{
 /
*************************************************************
 /* FPRoundInt converts a fixed point number, FPVal,
   with NbrFracBits to the right of the radix point
    into a long integer value. */
 /
*************************************************************

 /* round before we convert */
 if (FPVal < 0)
  FPVal = FPVal - (1 << (NbrFracBits-1));
 else
 FPVal = FPVal + (1 << (NbrFracBits-1));

 return (FPVal >> NbrFracBits);
}


static long __inline LongIntToFP(long IntVal,
                                 long NbrFracBits)
{
 /
*********************************************************
 /* LongIntToFP converts a long integer, IntVal, into a
   fixed point number with NbrFracBits to the right
   of the radix point. */
 /
*********************************************************

 return (IntVal << NbrFracBits);
}


static long __inline DoubleToFP(double DoubleVal,
                                long NbrFracBits)
{
 /
*********************************************************
 /* DoubleToFP converts a double (floating point) number,
   DoubleVal into a fixed point number with NbrFracBits
   to the right of the radix point. */
 /
*********************************************************

 /* Round before conversion */
 DoubleVal = DoubleVal * (1 << NbrFracBits);
 if (DoubleVal < 0.0)
  DoubleVal = DoubleVal - 0.5;
 else
  DoubleVal = DoubleVal + 0.5;

 return ((long)DoubleVal);
}


long RLPm2Pts (double t0, double x0,
               double t1, double x1,
               double *b, double *c,
               long *RL0, long *DeltaRL)
{
 /
*********************************************************
 /* RLPm2Pts computes the coefficients, b and c,
   for the equation of a line, x = b * t + c, given two
   points, (t0, x0) and (t1, x1), on that line. It also
   computes the initial values used by the
   function RL described in the article. RL0 and
   DeltaRL are fixed point numbers. The function returns
   the rounded integer portion of RL0 to the caller. */
 /
*********************************************************

 /* Compute b and c in x = b * t + c */
 *b = (x1 - x0) / (t1 - t0);
 *c = x0 - *b * t0;

 /* Compute RL0 and DeltaRL as fixed point numbers */
 *RL0 = DoubleToFP(x0, LIN_FRAC_BITS);
 *DeltaRL = DoubleToFP(*b, LIN_FRAC_BITS);

 /* return the integer portion of RL(0) */
 return (FPToLongInt(*RL0, LIN_FRAC_BITS));
}


long RQPm2PtsAndVel (double t0, double x0,
                     double t1, double x1,
                     double tm, double vm,
                     double *a, double *b, double *c,
                     long *RQ0, long *DeltaRQ0,
                     long *Delta2RQ)
{
 /
*********************************************************
 /* RQPm2PtsAndVel computes the coefficients, a, b,
   and c, for the equation of a parabola,
   x = a * t * t + b * t + c, given two
   points, (t0, x0) and (t1, x1), on the parabola and the
   slope, vm, at any point, (tm, xm) on the parabola.
   It also computes the initial values used by the
   function RQ described in the article. RQ0,
   DeltaRQ0, and Delta2RQ are fixed point numbers. The
   function returns the rounded integer portion of RQ0
   to the caller. */
 /
*********************************************************

 /* First compute a, b, and c. */
 *a = ((x1 - x0) - (vm * (t1 - t0))) /
      ((t1 * t1 - t0 * t0) - (2 * tm * (t1 - t0)));

 *b = vm - (2 * *a * tm);

 *c = x0 - ((*a * t0 * t0) + (*b * t0));

 /* Now compute RQ0, DeltaRQ0, and Delta2RQ */
 *RQ0 = DoubleToFP(x0, LIN_FRAC_BITS);
 *DeltaRQ0 = DoubleToFP( (*a * ((2 * t0) + 1)) + *b,
                        LIN_FRAC_BITS);
 *Delta2RQ = DoubleToFP(2 * *a, LIN_FRAC_BITS);

 /* return the integer portion of RQ(0) */
 return (FPToLongInt (*RQ0, LIN_FRAC_BITS));
}


long RLPm2Pts (double t0, double x0,
               double t1, double x1,
```

THIS LISTING IS NOT COMPLETE AS SHOWN

Complete source code and listings can be found on the AC's TECH disk.

Please write to:
Robert Galka
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722

TRUE BASIC

INPUT MASK WITH HELP KEY

BY T. DARREL WESTBROOK

I spend most of my programming time "human proofing" my programs. "Human proofing" requires program code that constantly checks user input ensuring that it is in proper format and within a specific range of values. This process uses my most valuable resource, my time. For an input error, the program guides the user to correct the input. Flexible program input requires more program coding to catch errors before they crash the program or invalidate calculations or output.

This article will explain a True BASIC design, which permits control over user input by using a keyboard mask. A help key is an integral part of the keyboard mask, and discussion of its capability will close this article. I used a True BASIC module to set up and define global variables used throughout the program. This is a powerful programming tool. Module discussion will cover specific structure considerations and practical examples. The term "procedure" in this article includes the True BASIC programming structures of subroutines, functions, and pictures. Italicized words are variables and the capitalized words are True BASIC keywords. Line numbers are for reference and are not necessary for program operation.

True BASIC is a structured, portable language. The program will run, with minor graphics mode changes, on an IBM-compatible or Macintosh computer. You must change line 36 when you port the program over to the different systems. It is high8 for an Amiga, COLOR for a Macintosh, and HIRES for an IBM compatible computer. This is true portability.

A True BASIC module has characteristics not shared by True BASIC functions, subroutines, or libraries. True BASIC scans each module at program startup and initializes appropriate code before main program execution. Functions and subroutines do not normally share variable values or names outside the procedure. These variables are local values to the procedure. Consider the output of the following example.

```
CALL Alpha
END
EXTERNAL
SUB Alpha
            LET My_Variable = 5
            CALL Bravo
            PRINT My_Variable
END SUBSUB Bravo
            LET My Variable = 10
END SUB
```

When executed, this program will print the number five to the screen. True BASIC considers My_Variable, within each subroutine, as two different, unrelated variables. If you wanted Alpha and Bravo to share My_Variable, you have three options. One is by reference, and the last two options use modules and the keywords SHARE and PUBLIC. I'll cover the reference option first.

To modify the preceding code to pass the value of My_Variable by reference the code would look like:

```
CALL Alpha
ENDEXTERNAL
SUB Alpha
            LET My_Variable = 5
            CALL Bravo(My_Variable)
            PRINT My_Variable
END SUB
SUB Bravo(Passed_Data)
                        LET Passed_Data = 10
END SUB
```

The result of this code execution is the number 10 printed on the screen. The Passed_Data variable in subroutine Bravo takes on the name and value of the My_Variable. Any changes made to Passed_Data in subroutine Bravo will effect My_Variable in subroutine Alpha. You lose the value of My_Variable when program flow returns to Main. The other options for passing data to called procedures use modules and the SHARE and PUBLIC keywords.

You use the SHARE keyword in the module header. SHARE allows procedures to use variables, arrays, and pictures throughout the module. The SHAREd item retains its assigned value even after program flow exits from the module. Consider the following code:

```
CALL Alpha
END
EXTERNAL
MODULE Test
SHARE My_Variable
SUB Alpha
            LET MY_Variable = 5
            CALL Bravo
            Print My_Variable
END SUB
SUB Bravo
            LET MY_Variable = 10END SUBEND MODULE
```

It has the same effect as passing My_Variable values by reference, but I eliminated (My_Variable) and (Passed_Data) from the code. It has the added benefit of returning its assigned value when program flow exits the module. Another example is the SHARE #99 statement on line 34. Module specific variables, which retain their values, create a very flexible programming capability. The module PUBLIC keyword is the last option which allows variable sharing within a program.

To establish a public variable, use the True BASIC keyword PUBLIC. You declare a variable PUBLIC once and it must be PUBLIC before assigned any value. If you made the following PUBLIC statement in subroutine Alpha,

```
SUB Alpha
          PUBLIC My_Global_Variable
          (subroutine body)
END SUB
```

and then tried to make a similar PUBLIC statement in subroutine Bravo:

```
SUB Bravo
  PUBLIC My_Global_Variable
   (subroutine body)
END SUB
```

you get a runtime error stating that the variable "Name can't be redefined." If My_Global_Variable is assign a value before the PUBLIC statement, the same runtime error will occur. The DIM keyword is the normal method for dimensioning a matrix or array. However, when an array is a public variable, it is dimensioned and made public with one statement (see listing, line 30). I locate my public variables in a module named Global to avoid this problem. I have one place where I add, change, or delete public variables. There are two other key words used in the module header, PRIVATE and DECLARE DEF.

The PRIVATE keyword prevents the named procedure from access outside the module. Only procedures in a module can use a PRIVATE variable or procedure within the module. In the following example, subroutine Bravo is PRIVATE. From inside the module Alpha uses Bravo, but Bravo is not usable from the Main program. The following code demonstrates this idea:

```
CALL Bravo
END
EXTERNAL
MODULE Test
SHARE My_Variable
PRIVATE Bravo
SUB Alpha
          LET My_Variable = 5
          CALL Bravo
          PRINT My_Variable
END SUBSUB Bravo
          Let My_Variable = 10
END SUB
END MODULE
```

This results in the error, "Undefined routine Bravo in Main Program," which shows that Bravo is not usable from outside the module. If you add the following code lines immediately following 'END MODULE',

```
SUB Bravo
          Let My_Variable = 10
END SUB
```

then run the program, True BASIC will report "You have two routines called test bravo." The use of 'test' in the error statement is to direct you to a module named 'test'. Within test there is a subroutine named bravo which is a duplicate name for another subroutine located outside the module. The Help_Screen PICTURE is an example of the

```
HelpKey = 0

12345
```

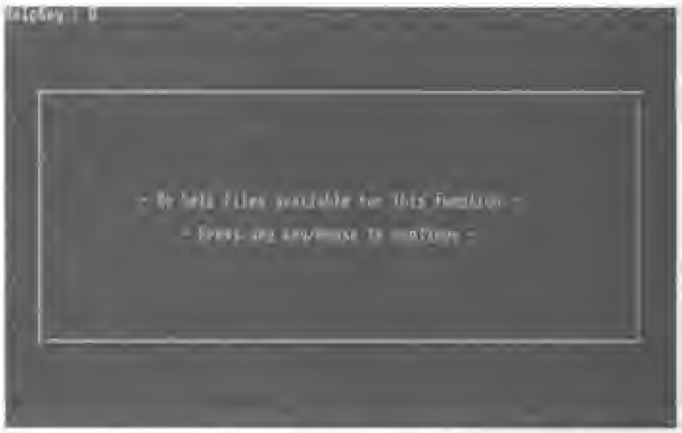
PRIVATE statement usage (lines 65 through 98).

Access to Help_Screen PICTURE is in the module and it is hidden from the rest of the program. DECLARE DEF is the last True BASIC keyword used in the module header. Since the program listing does not contain a function, I'll use an imaginary function called cdate$ to illustrate its use within a module.

DECLARE DEF statement informs True BASIC that the program intends to use a function within the current procedure. When used in a module header, True BASIC permits function use by any procedure within the module. Suppose a function called cdate$ returns the computer system date in a format specified by a string (format$). Assume the string is a combination of Ds (for day), Ms (for month), and Ys (for year). The position of the Ds, Ms, and Ys determines the format of the returned date string. For example, "DDMMMYY" will return the date as "15 Jul 93."

The following code illustrates the use of DECLARE DEF without modules.

```
CALL Alpha
CALL Bravo
ENDEXTERNAL
SUB Alpha
          DECLARE DEF cdate$
          PRINT cdate$("DDMMMYY")
END SUB
SUB Bravo
          DECLARE DEF cdate$
          PRINT cdate$("DDMMMYY")
END SUB
DEF cdate$(format$)
          (conversion code)
          LET cdate$ = "15 Jul 93"
END DEF
```



```
HelpKey : 0

- No help files available for this function -
- Press any key/mouse to continue -
```

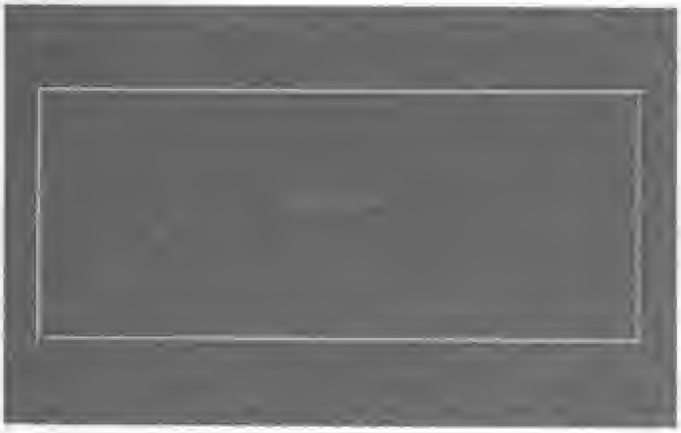
Workbench Window Icons Tools

Backdrop

Execute Command

Amazing Computing

What's the best way to improve productivity on your Workbench?



Notice the DECLARE DEF statement in each of the subroutines. This can get cumbersome if many procedures use the function. When subroutines are in a module, you use the DECLARE DEF cdate$ once and all procedures within the module can use the function. Modify the previous code as follows:

```
CALL Alpha
          CALL Bravo
          EXTERNAL
          MODULE Test
          DECLARE DEF cdate$
          SUB Alpha
                    PRINT cdate$("DUMMY")
          END SUB
          SUB Bravo
                    PRINT cdate$("DUMMY")
          END SUB
          END MODULE
          DEF cdate$(format$)
                    (conversion code)
                    LET cdate$ = "15 Jul 91"
          END DEF
```

which shows how to use DECLARE DEF in a module.

The Global module in the program listing uses PUBLIC, PRIVATE, and SHARE key words. Now I'll explain the Help Key subroutine and its use within a program.

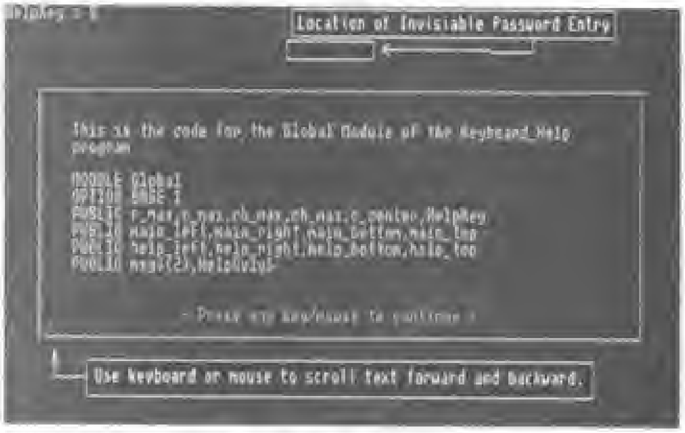
The keyboard input mask is a module named Input_Mask. Three subroutines, Keyboard, Keyboard_Center, and Keyboard_Single compose the module (see lines 171 to 410). Keyboard_Single is PRIVATE and not accessible outside of the module. It is the entry point for the Help Key procedure. Keyboard and Keyboard_Center both use Keyboard_Single to control program input. You use Keyboard to enter data anywhere on the screen and Keyboard_Center will center the input on a line.

Keyboard call format is:

```
                    CALL Keyboard(r,c,m,c$,PC,IS,CS,CC)
          where:
                    - r and c are screen row and column to start keyboard
input,
                    - m is the maximum desired character length of the input.
                    - c$ is a string of printable characters,
                    - PC is the pen color,
                    - IS is the input select, which determines acceptable
keyboard characters,
                    - CS is cursor set (ON or OFF 1 or 0), and
                    - CC is cursor color.
```

Keyboard_Center uses an additional parameter named Password_Set (either 1 or 0), which protects input data without displaying it on the screen. Since row and column (row1 and col1) variables change during the execution of Keyboard, working variables, row and col, are assigned their values. Lines 174 to 182 are self explanatory. Lines 183 to 189 empty the keyboard buffer. This prevents unwanted characters from getting into the input stream. If you fill up the keyboard buffer by holding down any key, the subroutine will not progress past these lines until you release the pressed key. The Buffer (line 147) subroutine could replace lines 183 to 189, which would also clear the keyboard and mouse inputs. After the keyboard buffer is clear, the Keyboard subroutine calls Keyboard_Single (line 198).

Keyboard_Single is the workhorse of the input mask. It limits the keyboard input stream, it is the help key access, and it controls a large portion of the module's "human proofing." GET KEY keycode takes


Display = 1
Location of Invisiable Password Entry
- Press any key/mouse to continue -
Use keyboard or mouse to scroll text forward and backward.

```
This is the code for the Global Module of the Keyboard_Help
program

MODULE Global
OPTION BASE 1
PUBLIC r_max,c_max,ch_max,rh_max,c_center,HelpKey
PUBLIC main_left,main_right,main_bottom,main_top
PUBLIC help_left,help_right,help_bottom,help_top
PUBLIC msg$(2),HelpOnly$
```

the next input from the keyboard and assigns it the value of keycode. This is an integer and it represents the ASCII code of the key you pressed. When you press the "A" key, keycode would be 65. You can find the ASCII character set in the *True BASIC Reference Manual*, page 375, or in the *Student Edition Manual*, page 141. The following program will display the keycode of pressed keys, if you don't have the manuals handy.

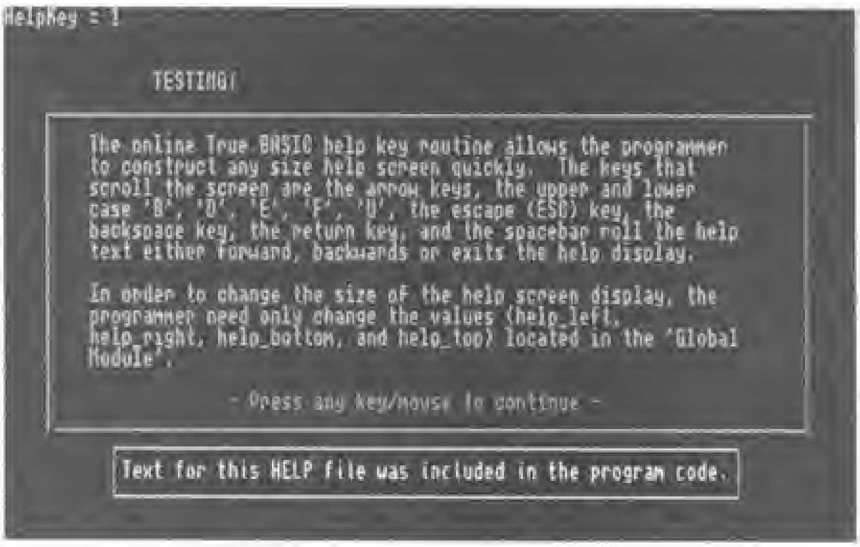
```
DO
   GET KEY keycode
   SET cursor 12,40
   PRINT using$("###",keycode)
   PAUSE 5 ! waits five seconds
LOOP
END
```

Once keycode has a value, the program determines if it is the keyboard HELP key. When keycode is 325, the subroutine branches to the Help_Display subroutine (lines 412 to 639). I will cover more of the help key selection in a moment. When keycode is not 325, program flow continues with the SELECT CASE options (lines 314 to 401).

The CASE structure operates much like an IF...THEN statement, but you can check more keycode values with one CASE statement than is possible for a single IF...THEN structure. The Keyboard_Single subroutine is easily customized for each program use.

The value assigned Input_Select determines which CASE selection monitors keyboard input. If a portion of my program needed numbers as inputs, then I would assign Input_Select equal two. This limits acceptable input to the zero through nine keys, the backspace key, the return key, and the delete key. Press any other key and the program appears to do nothing. The actual program flow branches to the Input_Select CASE 2 (line 354) determines it was not acceptable input, and then branches back to line 310 to wait for the next key stroke. There is no discernible time delay while typing in the data. Now let's discuss what happens when you select the HELP key.

When program flow is in the Keyboard_Single subroutine and you select the HELP key, the program branches to the Help_Display subroutine (lines 412 through 639). HelpKey, a global variable, is normally equal to zero, but it can change anytime during program flow. The best place to set the HelpKey is entering and exiting the using procedure. There are three HelpKey CASE examples shown in the program listing. They are CASE 1 (text$ is in the program code), CASE 2 (text$ taken from a text file), and CASE ELSE (when HelpKey is zero). You can add any number of CASE options in your program help procedure.


```
HelpKey = 1

        TESTING!

    The online True BASIC help key routine allows the programmer
    to construct any size help screen quickly.  The keys that
    scroll the screen are the arrow keys, the upper and lower
    case 'B', 'D', 'E', 'F', 'U', the escape (ESC) key, the
    backspace key, the return key, and the spacebar roll the help
    text either forward, backwards or exits the help display.

    In order to change the size of the help screen display, the
    programmer need only change the values (help_left,
    help_right, help_bottom, and help_top) located in the 'Global
    Module'.

              - Press any key/mouse to continue -

  Text for this HELP file was included in the program code.
```

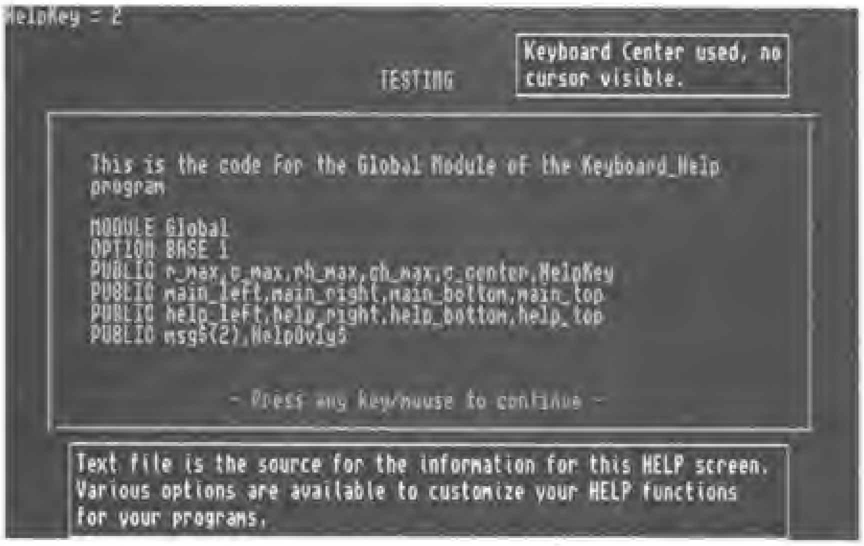
The DECLARE PUBLIC statement in Help_Display permits the use of global variables within the subroutine. I used the s$ and t$ arrays to load and manipulate the text$ data embedded within the program code (see lines 440 to 443) or read from a text file (see lines 446 to 461).

The program breaks the text$ data into paragraphs and places the data in the s$ array (i.e., each element in the array is a complete paragraph), and each line of the t$ array represents a single line that will print within the help screen dimensions. The text information in the t$ array allows forward and backward browsing of text$.

I used three blanks in a row to inform the program code where to make the paragraph breaks. At the end of line 442, immediately after '... the help display' are three blanks. This causes the subroutine to insert a blank line in the s$ array. This effectively causes a blank line to appear after each paragraph. Figure 1 is an example of the text$ variable outputted to the help screen. The next example, CASE 2 (line 445), will read the help information from a text file. I used a file named "Key_CASE_2.txt," which has 84 lines of program code. The length of the text file affect how long it takes to read the data. Each text file line represents a paragraph when the program loads the information into the s$ array. I used the True BASIC editor to make and save the text file.

There are other variations you could make with the Help_Display subroutine: different size and placement of the help screen, loading HelpOvly$ from a byte file rather than drawing it initially, and popup info screens to help guide data entry, to suggest a few. Each of us has ideas that will make these procedures beneficial for all to add to their True BASIC library.

You can use the True BASIC keyboard mask, with built-in help key capability, to write programs quickly. It will minimize your "human proofing" time, permitting you to spend your most precious resource on program design and operation.


```
HelpKey = 2
                                          Keyboard Center used, no
                               TESTING    cursor visible.

    This is the code for the Global Module of the Keyboard_Help
    program

    MODULE Global
    OPTION BASE 1
    PUBLIC r_max,c_max,rh_max,ch_max,c_center,HelpKey
    PUBLIC main_left,main_right,main_bottom,main_top
    PUBLIC help_left,help_right,help_bottom,help_top
    PUBLIC msg$(2),HelpOvly$

              - Press any key/mouse to continue -

  Text file is the source for the information for this HELP screen.
  Various options are available to customize your HELP functions
  for your programs.
```

Program Listing

```
!
! 'Key_Help' copyrighted
!  by T. Darrel Westbrook
!
1 DECLARE PUBLIC rh_max,ch_max,Helpovly$,HelpKey
2 DECLARE PUBLIC help_left,help_right,help_bottom,help_top
3 DO
4     CLEAR
5    PRINT "HelpKey = 0"
6     LET HelpKey = 0
7   CALL Keyboard(10,49,10,c5,2,1,1,3)  ! only numbers
8     CLEAR
9    PRINT "HelpKey = 1"
10     LET HelpKey = 1
11    CALL Keyboard(4,15,10,c5,2,0,1,1)  ! accept any key
12     CLEAR
13     PRINT "HelpKey = 2"
14     LET HelpKey = 2
15   CALL Keyboard_Center(4,10,c5,2,4,1,2,4)  ! every-
thing upper case
16     CLEAR
17     PRINT "HelpKey = 3"
18     LET HelpKey = 3
19   CALL Keyboard_Center(12,10,c5,2,1,2,3,1)
20 LOOP
21 END
22
23 EXTERNAL
24
25 MODULE Global
26 OPTION BASE 1
27 PUBLIC r_max,c_max,rh_max,ch_max,c_center,HelpKey
28 PUBLIC main_left,main_right,main_bottom,main_top
29 PUBLIC help_left,help_right,help_bottom,help_top
30 PUBLIC msg$(2),Helpovly$
31
32 PRIVATE Help_Screen ! call Help_Screen in module
33
34 SHARE #99  ! shared channel within module
35
36 SET MODE "high8"
37 SET color mix (0) 0.5,1,0.6         ! background light
blue
38 SET color mix  (1) .7,.7,.7          ! set color to white
39 SET color mix  (2) 0,1,0             ! set color green
40 SET color mix  (3) 1,0.1,0.7         ! set color orange
41 SET color mix  (4) 1,1,0             ! set color yellow
42 SET color mix  (5) 1,0,0             ! set color red
43 SET color mix  (6) 0,0,.45           ! set color dark blue
44 SET color mix  (7) 0,0,0             ! set color black
45
46 ASK max cursor r_max,c_max
47 LET c_center = c_max ! center string on current
window
48 LET HelpKey = 1  ! initialize help key code
49 ASK screen main_left,main_right,main_bottom,main_top
50
51 LET msg$(1) = "- Press any key/mouse to continue -"
52 LET msg$(2) = "- Press any key to continue -"
53
54 DRAW Help_Screen  ! initial showing of help screen
55 WINDOW #99  ! activate help screen
56
57 BOX SHOW Helpovly$ at 0,0 ! actual help screen display
58 LET c_center = ch_max  ! column width to help screen
size
59 CALL Center("- Initializing -",ch_max/2,1,5)
60 PAUSE 2  ! pause at help screen
61 CLEAR
62 CLOSE #99  ! close the help screen
63 LET c_center = c_max  ! column width to original screen
64
65 PICTURE Help_Screen
66     LET help_left = 0.05
67     LET help_right = 0.95
68     LET help_bottom = 0.2
69     LET help_top = 0.9
70   OPEN #99 : screen help_left,help_right,help_bottom,
help_top
71     WINDOW #99
72    ASK max cursor rh_max,ch_max
73     SET color 5
74   BOX LINES 0.005,0.995,0.01,0.99
75   BOX LINES 0.007,0.993,0.005,0.995
76   BOX LINES 0.009,0.991,0.005,0.995
77     SET color 1
78     PLOT LINES : 0,0; 0,1 ; 1,1
79    PLOT LINES : 0.0025,0.008; 0.0025,1
80    PLOT LINES : 0.002,0.005; 0.002,1
81     SET color 7
82    PLOT LINES : 0.002,0 ; 1,0; 1,0.99
83    PLOT LINES : 0.998,0.99; 0.998,0
84    PLOT LINES : 0.996,0; 0.996,0.99
85      !
86    ! draw the inside highlight
87      !
88    PLOT LINES : 0.011,0.015 ; 0.011,0.985
89    PLOT LINES : 0.013,0.985 ; 0.011,0.015
90    PLOT LINES : 0.989,0.985 ; 0.011,0.985
91     SET color 7
92    PLOT LINES : 0.989,0.015 ; 0.989,0.975
93    PLOT LINES : 0.989,0.017 ; 0.011,0.017
94    PLOT LINES : 0.987,0.015 ; 0.015,0.015
95     SET color 2
96   BOX LINES 0.009,0.991,0.005,0.995
97    BOX KEEP 0,1,0,1 in Helpovly$
98 END PICTURE  ! end of 'PICTURE Helpovly'
99 !
100 SUB Open_Help(#9)
101     OPEN #9 : screen
help_left,help_right,help_bottom,help_top
102     WINDOW #9
103 END SUB
104 END MODULE  ! end of 'Global'
105
106 SUB Press_Any(row,Pen_Color)
107     DECLARE PUBLIC msg$()
108    CALL Center(msg$(1),row,Pen_Color)
109     CALL Press_On
110 END SUB  ! end of 'Press_Any'
111
112 SUB Press_On
113     CALL Buffer ! clears the keyboard and mouse
buttons
114     LET w = 0  ! mouse left button state
115     DO  ! now take any input to continue
116         if key input then
```

```
117          EXIT SUB  ! some key pressed
118       ELSE
119          GET MOUSE x,y,s
120          IF s = 2 then EXIT SUB  ! mouse button
pressed
121       END IF
122    LOOP
123 END SUB  ! end of [illegible]
124
125 SUB Center(text$,row,Pen_Color)
126    DECLARE PUBLIC r_max,c_center
127    IF row>r_max then LET row = r_max-1
128    ASK color hold  ! remember the current text color
129    SET color Pen_Color  ! change text color
130    SET cursor row,int(c_center-(len(trim$(text$)))/2)
131    PRINT text$
132    SET color hold  ! restore text color
133 END SUB  ! end of 'Center'
134
135 SUB Erase_Line(row_1,row_2)
136    DECLARE PUBLIC r_max,c_max
137    IF row_1 > row_2 then LET row_1 = row_2
138    IF row_1 <= 0 then LET row_1 = 1
139    IF row_2 > r_max then LET row_2 = r_max
140    LET b$ = repeat$(" ",c_max-1)  ! clear to end of
screen
141    FOR r=row_1 to row_2
142       SET cursor r,1
143       PRINT b$
144    NEXT r
145 END SUB  ! end of 'Erase_Line'
146
147 SUB Wait_key
148    SET cursor "OFF"
149    DO  ! clear keyboard buffer
150       IF key input then
151          GET KEY b  ! get any keyboard input
152          GET MOUSE x,y,s  ! get mouse input too
153       ELSE
154          GET MOUSE x,y,s
155          IF s <> 0 and NOT key input then EXIT SUB
156       END IF
157    LOOP
158 END SUB  ! end of 'Wait_key'
159
160 SUB Space_Finder(string$,Start,End,insert)
161    LET insert = 0  ! error, could not find a blank
162    FOR n=Start to Start-End step -1  ! first occurrence of "
"
163       !                                 from right to left
164       IF mid$(string$,n,1) = " " then  ! blank was found
165          LET insert = n
166          EXIT FOR
167       END IF
168    NEXT n
169 END SUB  ! end of 'Space_Finder'
170
171 MODULE Input_Mods
172 PRIVATE Keyboard_Single
173 SUB
Keyboard(row1,col1,space,c$,Pen_Color,Input_Select,
Cursor_Set,Cursor_Color)
174    SET cursor "OFF"
175    LET row = row1  ! don't use original variable
176    LET col = col1  ! so row and col will not change
value
177    ASK color holdcolor
178    LET colmin = col  ! remember the starting column
179    SET cursor row,col
180    PRINT repeat$(" ",(space+1))  ! clear box where
text will display
181    SET cursor row,col
182    LET c$ = ""  ! initialize text string to null
183    DO  ! forever loop
184       IF key input then  ! something in keyboard
buffer
185          GET KEY x  ! then get it
186       ELSE
187          EXIT DO  ! buffer clear
188       END IF
189    LOOP  ! end of forever loop to clear keyboard
190    DO  ! forever loop
191       IF Cursor_Set = 1 then  ! i.e., display cursor
192          SET color Cursor_Color
193          SET cursor row,col
194          PRINT "|";  ! cursor character
195       END IF  ! end of 'IF Cursor_Set = 1 ...'
196       SET cursor row,col
197       SET color Pen_Color
198       CALL Keyboard_Single(keycode,f$,Input_Select)
199       SELECT CASE keycode
200       CASE 8, 127  ! backspace, 'DEL' key
201             LET col = col - 1
202             SET cursor row,col
203             PRINT repeat$(" ",(colmin + space + 1 -
col))
204             SET cursor row,col
205             LET c$ = c$[1:len(c$)-1]  ! take off last
input letter
206             IF col < colmin then
207                SOUND 150,.15  ! sound bell note if
backspace too far left
208                LET col = colmin  ! let col = colmin
209             END IF
210       CASE 13  ! CR
211             WHEN error in
212                IF ord(c$) = -1 and len(c$) = 0 then
LET c$ = ""
213             USE
214                ! no error, just return c$ as null
215             END WHEN
216             EXIT DO  ! exit for CR
217       CASE 32 to 126  ! printable codes
218             SET color Pen_Color
219             SET cursor row,col
220             LET c$ = c$ & f$  ! add new printable
character to string
221             PRINT f$  ! print the character
222             LET col = col + 1  ! increment col counter
223             IF len(c$) >= space then EXIT DO  ! no
more input than designed
224       CASE else
225             ! no exit, and don't change a thing
226       END SELECT
227    LOOP  ! end of forever loop
228    LET c$ = trim$(c$)  ! trim off all blanks
229    SET cursor row,colmin  ! return cursor to starting
position
230    SET color holdcolor  ! return to original color
231 END SUB  ! end of 'Keyboard'
232
233 SUB Keyboard_Center(row1,space,c$,Pen_Color,
Input_Select,Cursor_Set,Cursor_Color,Password_Set)
234    DECLARE PUBLIC c_max
```

```
   LET row = row1
   DO ! forever loop
      IF key input then ! clear the keyboard buffer
         GET KEY z
      ELSE
         EXIT DO
      END IF
   LOOP ! end of forever loop
   LET col = c_max/2 ! start at center of the screen
   SET cursor "OFF"
   ASK color HoldColor ! remember original color number
   SET cursor row,(c_max/2)-(space+2)/2
   PRINT repeat$(" ",space+2) ! clear the input box
   LET c$ = "" ! initialize return string, null
   DO ! forever loop
      IF Cursor_Set = 1 then ! if 1 then turn on
         SET color Cursor_Color ! set cursor color
         SET cursor row,int((c_max-len(c$))/2) + len(c$)
         PRINT "|"
      END IF
      CALL Keyboard_Single(keycode,t$,Input_Select)
      SELECT CASE keycode
      CASE 8, 127 ! BS and DEL keycode
           IF col < 40 then
              SOUND 150,.15 ! sound bell tone if backspace too far
           ELSE
              LET c$ = c$[1:len(c$)-1] ! remove last input to string
              SET color Pen_Color      ! Pen color
              SET cursor row,col
              PRINT repeat$(" ",len(c$) + 1)
              SET cursor row,int((c_max-len(trim$(c$)))/2) ! center the current string
              IF Password_Set=1 then ! password protect
                 PRINT repeat$("*",len(c$)) & " "
              ELSE
                 PRINT c$ & " "
              END IF
              LET col = col - 1 ! decrement column
           END IF
      CASE 13 ! CR
           WHEN error in
              IF ord(t$) = -1 and len(c$) = 0 then LET c$ = ""
           USE
              ! don't exit program
           END WHEN
           EXIT DO ! Exit if character is CR
      CASE 32 to 126 ! printable characters
           SET color Pen_Color
           SET cursor row,col + 1
           LET c$ = c$ & t$ ! add new printable character
           SET cursor row,int((c_max-len(trim$(c$)))/2)
           IF Password_Set = 1 then ! password protect
              PRINT repeat$("*",len(c$))
           ELSE
              PRINT c$
           END IF
           IF len(c$) = space then EXIT DO ! c$ is equal to the string length
              LET col = col + 1 ! increment col counter
      CASE else
              ! selected which are allowed by program
      END SELECT
   LOOP ! End of forever loop
   SET color Pen_Color
   SET cursor row,int((c_max-len(c$))/2)
   IF Password_Set = 1 then ! returning password
      PRINT repeat$("*",len(c$)) & " "
   ELSE
      PRINT c$ & " "
   END IF
   LET c$ = trim$(c$)
   SET color HoldColor ! return original pen color
END SUB ! end of 'Keyboard_Center'

SUB Keyboard_Single(keycode,t$,Input_Select)
    DECLARE PUBLIC HelpKey
    DO ! First order loop
       GET KEY keycode
       IF keycode = 325 then ! help key entered
          CALL Help_Display ! go to help screen routine
       ELSE
          SELECT CASE Input_Select
          CASE 0 ! normal input
               SELECT CASE keycode
               CASE 8,13,127 ! BS, CR, delete key
                    EXIT DO
               CASE 32,35,38,39 ! Space # & '
                    EXIT DO
               CASE 40,41,44,45,46,47 ! ( ) , - . /
                    EXIT DO
               CASE 48 to 57       ! numbers 0 through 9
                    EXIT DO
               CASE 58,60,61,62 ! : < = >
                    EXIT DO
               CASE 65 to 90       ! letters A through Z
                    EXIT DO
               CASE 97 to 122      ! letters a through z
                    !
                    ! *** erase the following line if you want lower case letters
                    LET keycode = keycode - 32 ! make everything upper case
                    EXIT DO
               CASE 315 to 324 ! function keys F1 to F10
                    EXIT DO
               CASE 325 ! HELP key
                    EXIT DO
               CASE 328,336,331,333 ! up, down, left, right arrow key
                    EXIT DO
               CASE 429,437,371,372 ! SHIFT up, down, left, right arrow key
                    EXIT DO
               CASE else
                    ! non-selectable, maintain loop
               END SELECT
          CASE 1 ! BS, CR, numbers & period only allowed
               SELECT CASE keycode
               CASE 8,13,46,127 ! BS, CR, period key,
delete key
                        EXIT DO
                CASE 48 to 57          ' numbers 0 through 9
                        EXIT DO
                CASE else
                        ' not selectable, remain in loop
                END SELECT
        CASE 2  ' numbers only allowed
                SELECT CASE keycode
                CASE 8,13,127          ' BS, CR, delete key
                        EXIT DO
                CASE 48 to 57          ' numbers 0 through 9
                        EXIT DO
                CASE else
                        ' not selectable, remain in loop
                END SELECT
        CASE 3 ' BS, CR, numbers & period only allowed
                SELECT CASE keycode
                CASE 8,13,45,46,127  ' BS, CR, minus key, period key, delete key
                        EXIT DO
                CASE 48 to 57          ' numbers 0 through 9
                        EXIT DO
                CASE else
                        ' not selectable, remain in loop
                END SELECT
            CASE 4
                SELECT CASE keycode
                CASE 8,13,127          ' BS, CR, delete key
                        EXIT DO
                CASE 32,45,46,48   ' space [illegible]
                        EXIT DO
                CASE 48 to 57          ' numbers 0 through 9
                        EXIT DO
                CASE 65 to 90          ' letters A through Z
                        EXIT DO
                CASE 97 to 122         ' letters a through z
                        '
                        '*** erase the following line if you want lower case letters
                        LET keycode = keycode - 32   ' make everything upper case
                        EXIT DO
                CASE else
                        ' not selectable, remain in loop
                END SELECT
        CASE 5 ' BS, CR, & numbers only allowed
                SELECT CASE keycode
                CASE 8,13,45,127 ' BS, CR, minus key, delete key
                        EXIT DO
                CASE 48 to 57          ' numbers 0 through 9
                        EXIT DO
                CASE else
                        ' not selectable, remain in loop
                END SELECT
        CASE else
                ' not selectable, remain in loop
        END SELECT
```

```
        END IF ' end of "IF keycode [illegible]"
    LOOP ' end of "input" loop
    IF keycode >= 32 and keycode <= 126 then
        PRINT CHR$(keycode); ' print visible character
    ELSE
        PRINT " "; ' print [illegible] character
    END IF
END SUB ' end of "keyboard_filter"
SUBROUTINE [illegible]
'
SUB help_display
   DECLARE SHARED c_max,r_max,c_center AS INTEGER, [illegible]
   DECLARE SHARED main_left,main_right,main_bottom, main_top
   DECLARE SHARED help_left,help_right,help_bottom, help_top
   DIM [illegible]
   '
   ' [illegible]
   '
   LET c_hold = c_center ' [illegible]
   LET c_center = c_max ' number of characters on the help display
   ASK [illegible] hold_row,hold_col ' [illegible] cursor position
   BOX KEEP main_left,main_right,main_bottom,main_top IN hold_screen$
   '
   ' clear the help screen
   '
   CLEAR AREA ... help_left,help_right, help_bottom,help_top
   WINDOW #0
   BOX [illegible] ' [illegible] the screen
   LET r_pos = int(r_max/2) ' the mid row of screen
   LET [illegible] ' [illegible]
   LET indent = 2 ! left margin
   '
   ' [illegible]
   ' [illegible] "CASE 1" example
   ' [illegible] from a text file [illegible] "CASE 2" example
   '
   SELECT CASE [illegible]
   CASE 1 ' help example with text in the program code
         LET text$ = "[illegible] help key routine allows the programmer to [illegible] help screen."
         LET text$ = text$ & "Quickly: The keys that scroll the screen are the arrow keys, the upper and lower case 'D', 'U', 'F', 'B', 'U', the escape (ESC) key, "
         LET text$ = text$ & "the backspace key, the return key, and the spacebar scroll the help text either forward, backwards or exits the help display. "
         LET text$ = text$ & "In order to change the size of the help screen display, the programmer need only change the values help_left, help_right, help_bottom, and help_top' located in the 'Global Module'."
         CALL [illegible]
   CASE 2 ' help with text taken from a text file
         OPEN #1 : name "[illegible].txt", [illegible]
```

```
443         LET text$ = "" ! initialize
444         LET n = 0
445         DO while more #1
446            LET n=n+1  ! count lines in [illegible]
447            LINE INPUT #1 : a$
448         LOOP
449         SET #1 : POINTER BEGIN
450       MAT a$ = nul$(n) ! redimension [illegible]
array
451         LET n = 0
452         DO while more #1
453            LINE INPUT #1 : a$
454            LET n=n+1  ! [illegible] array [illegible]
455            LET a$(n) = a$
456         LOOP
457         CLOSE #1
458         [illegible]
459     CASE else  ! [illegible]
460         LET text$ = "" [illegible]
[illegible]
461         CALL [illegible]
462         [illegible]
463         CALL [illegible] #1
464     [illegible]
465     CLOSE #1
466     [illegible]
display [illegible]
467     SET cursor [illegible]
468     LET [illegible]
469     CALL [illegible] ! clear buffer
470     EXIT SUB
471
472     SUB Display_Help_( )
473        LET start(n, count) = [illegible]
474        LET y$ = ""
475        LET z$ = text$ ! use z$ as working copy of
text$
481        IF text$[len(text$)-2:len(text$)] = "[illegible]" then
482           LET y$ = string$("*****",len(text$)-2) &
string$("*****",len(text$))
483           LET end = len(text$)-1  ! allow for the
paragraph marker at the end
484        ELSE
485           LET end = len(text$)
486        END IF
487        LET count=0  ! [illegible]
488        DO
489           LET x = pos(z$,[illegible],end)  ! [illegible]
[illegible]
490           IF x = 0 then  ! no more [illegible] in the
string
491              LET y$ = string$("*****",1) &
[illegible]("*****",end) & y$
492              EXIT DO
493           ELSEIF x = 1 then  ! paragraph marker at
start of string
494              LET count=[illegible]  ! [illegible]
[illegible]
495              LET y$ = [illegible]("*****",x) &
[illegible]("*****",end) & y$
496              LET y$ = [illegible]("*****",x) & [illegible]
[illegible]("*****",x) & y$
497              LET count = count + 1 ! increment
[illegible]
498              EXIT DO
499           ELSE                           ! x > 0
500              LET y$ = string$("*****",x-1) &
string$("*****",end) & y$
501              LET y$ = string$("*****",x) &
string$("*****",end) & y$
502              LET count = count + 1
503              LET end = x-1
504           END IF
505        LOOP
506        MAT y$ = [illegible]  ! [illegible]
[illegible]
507        LET k = 0  ! initialize array [illegible]
508        FOR n=1 to [illegible] step 2
509           LET k = k + 1
510           LET [illegible]
[illegible]
511        NEXT n
512        LET x = 0  ! [illegible]
513        MAT t$ = [illegible]_max-
indent*2)*count-k)  ! [illegible]
514        FOR n=1 to [illegible]  ! [illegible] array [illegible]
paragraph array
515           IF [illegible] then  ! [illegible]
[illegible] only
516              LET k = k + 1  ! increment [illegible]
[illegible]
517              LET [illegible]
518           ELSE  ! parse the paragraph
519              LET start = 1
520              DO while start <= len([illegible])
521                 LET k=k+1
522                 IF [illegible]_max_indent*2 >
len([illegible]) then
523                    LET t$(k) = [illegible]
len([illegible]))
524                    EXIT DO
525                 ELSE  ! [illegible]
526                    CALL [illegible]
start+(t_max-indent*2),x)
527                    IF [illegible]  ! [illegible]
[illegible]
528                       LET t$(k)=[illegible]
start+(t_max-indent*2))
529                       LET start = start + [illegible]
indent*2)+1
530                    [illegible]  ! [illegible]
[illegible]
531                       LET
[illegible]
532                       LET start=x+1
533                    END IF
534                 END IF  ! end of "[illegible]_max
[illegible]..."
535              LOOP while start <= len([illegible])
536           END IF  ! end of "IF [illegible]
537        NEXT n
538        LET max_start = [illegible]  ! [illegible]
[illegible]
539        IF max_start < 0 then LET max_start = 1
540        CALL Display_Text
541     END SUB  ! end of "Display_Help_"
542     SUB Display_Text_
543        !
544        ! used to display text [illegible] from a text file
545        !
546        LET [illegible]
547        FOR n=1 to [illegible]  ! [illegible]
[illegible]
548           IF [illegible] <= t_max-indent*2 then
```

```
548              LET k=k+1  ! count one line
549            ELSE
550              LET start = 1
551             DO while start <= len(a$(n))
552                LET k=k+1
553               LET start = start + ch_max-indent*2
+1  ! allow for edge margins
554             LOOP while start <= len(a$(n))
555            END IF
556        NEXT n
557       MAT t$ = nul$(k+6)  ! use a big line buffer
558        LET k = 0
559       FOR n=1 to size(a$)  ! load t$() array
560          IF len(a$(n)) <= ch_max-indent*2 then
561             LET k=k+1  ! count another line
562             LET t$(k) = a$(n)
563            ELSE
564              LET start = 1
565             DO while start <= len(a$(n))
566               LET x = posr(a$(n)(start:start+
(ch_max-indent*2)),"  ",start+(ch_max-indent*2))
567               LET k=k+1  ! count another line
568                IF x = 0 then
569                 LET t$(k)=a$(n)(start:
start+(ch_max-indent*2))
570                 LET start = start+(ch_max-
indent*2)+1
571                ELSE
572                 LET t$(k)=a$(n)(start:x-1) !
don't use the blank
573                  LET start = start + x
574                END IF
575             LOOP while start <= len(a$(n))
576           END IF  ! end of 'IF len(a$) <= ch_max
...'
577        NEXT n
578       LET max_start = k-inc+1  ! prevent scrolling
past end of t$()
579       IF max_start < 1 then LET max_start = 1
580       CALL Display_Text
581    END SUB  ! end of 'Display_Help_2'
582    SUB Display_Text
583        !
584        ! display the text
585        !
586       LET start = 1  ! initialize array counter
587       LET w = 0  ! initialize left mouse button
588       CALL Buffer  ! clear keyboard buffer
589       CALL Help_PRT_Screen
590        DO  ! get a valid help key
591          IF key input then
592             GET KEY x  ! when there is an input
593             SELECT CASE x
594            CASE 13,32,68,70,100,102,336  ! move
FWD., CR, space, D, F, d, f, dwn-arrow
595                 CALL Help_PRT_Screen
596                 LET start = start+inc
597                 IF start > max_start then LET start
= max_start
598            CASE 8,66,85,98,117,328  ! move BAK, BS,
B, U, b, u, up arrow
599                 LET start = start-inc  ! move t$()
element pointer back
600                 IF start < 1 then LET start = 1
601                 CALL Help_PRT_Screen
602            CASE 27,69,101,331,333  ! ESC, E, e, left
arrow & right arrow
603                      EXIT SUB
604                CASE else
605                 ! Not a good selection, no exit
606                END SELECT
607             ELSE  ! check for mouse input
608               GET MOUSE u,v,w
609             LET r_test =ch_max-round(v*(ch_max-1)) !
converts screen coordinates
610              IF r_test > ch_max then LET r_test =
ch_max
611              IF r_test = 1 and w_test = 1 and then !
top half selected
612                 IF flag = 0 then LET start = start-
inc
613                 LET start = start-inc  ! move t$()
element pointer back
614                 IF start<1 then LET start = 1 !
(where start >= 1
615                 CALL Help_PRT_Screen
616                  LET flag = 1
617              ELSEIF r_test > 1 and and r_test <=
ch_max then
618                  IF flag = 1 then LET start =
start+inc
619                 CALL Help_PRT_Screen
620                 LET start = start+inc
621                 IF start > max_start then LET start =
max_start
622                  LET flag = 0
623              END IF  ! end of "IF r_test = 1 ..."
624           END IF  ! end of "IF key input ..."
625         LOOP
626    END SUB  ! end of 'Display_Text'
627    SUB Help_PRT_Screen
628        BOX SHOW helpcopy$ at 0,0  ! erase current
window
629        LET j = 1  ! display line counter
630        FOR line=start to (start+inc-1)
631            LET j = j + 1  ! display line counter
632            SET cursor j,indent  ! left margin
633            IF line > size(t$) then EXIT FOR
634            LET a$ = t$(line) & repeat$(" ",ch_max)
! pad the text
635            PRINT a$[1:(ch_max-indent*2)]  ! print the
line
636        NEXT line
637      CALL Prompt_Any(ch_max+1,1)
638    END SUB  ! end of 'Help_PRT_Screen'
639 END SUB  ! end of 'Help_Display'
```

Complete source code can be found on the AC's TECH disk.

Please write to:
T. Darrel Westbrook
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722

Mr. Newton And His Roots

Programming the Amiga in Assembly Language

By William P. Nee

Bye-bye

I said "good-bye" to an old friend yesterday—a friend who had helped me with assembly language programming for several years, a friend who got me through all those hours of confusion, GURU, and frustration. I finally packed up my Amiga 500. This article is written for the Amiga 3000, but, since the assembly code does not use any 68020/030 or co-processor commands it may still work on an Amiga 500 with some modifications.

In this article I'll discuss using Requesters. Previous articles have covered Menus (V3.1) and Gadgets (V3.2), since Requesters use Gadgets, this seems like a good time to learn about them. The assembly language program will solve equations and show the result graphically, and it uses requesters with a zoom routine and a picture-size control. I'll also show you how to have your program react to pressing specific keys rather than using menu selections.

Requesters

As with most Amiga features, Requesters are structured. In Table 1 I've listed the elements of the 112-byte requester structure. The OlderRequest is supplied by Intuition. Left-edge and top-edge refer to the number of pixels relative to the upper-left corner of the window. Width and height are the total requester dimensions in pixels. Relleft and Reltop refer to distances from the pointer if you want to position your gadget that way. The next three items are the pointers to the gadget structure, an optional border structure, and text structure for the gadget wording. Every requester must contain at least one gadget to escape from it and will usually contain two.

TABLE I
REQUESTER STRUCTURE (112 BYTES)

BYTE	
0	olderrequest - previous requesters (I)
4	left-edge - relative to upper-left corner of window
6	top-edge - relative to upper-left corner of window
8	width of requester in pixels
10	height of requester in pixels
12	relleft - location relative to the pointer
14	reltop - location relative to the pointer
16	reqgadget - pointer to the gadget structure
20	reqborder - pointer to an optional border structure
24	reqtext - pointer to intuitext structure
28	flags - ($1 pointrel - requester will be relative to the pointer) ($2 predrawn - there will be a custom bitmap structure)
30	backfill - inside pen color
32	reqlayer - pointer to layer structure
36	padding (32 bytes)
68	imagebmap - pointer to custom bitmap
72	reqwindow - pointer to requester window
76	padding (36 bytes)

Next are the flags for a requester. There are five of them:

```
POINTREL($1) - the requester will be positioned relative to the pointer (see
RELLEFT and RELTOP above)

PREDRAWN($2) - you will use a custom bitmap to draw your own requester

REQOFFWINDOW($1000) - the requester is active but off the window(maintained by
Intuition)

REQACTIVE($2000) - the requester is currently active (maintained by Intuition)

SYSREQUEST - this is a system-generated requester (maintained
by Intuition)
```

The background pen color is next followed by an optional Layer pointer. After 32 bytes of padding are the pointers to your optional bitmap and the window. More padding of 36 bytes finishes the structure.

An Alternative

Now that takes a lot of work for such a simple feature. Fortunately the Amiga has an easier way using Intuition's AutoRequest (-348 offset). Table II lists the items required to call this function. The requester will center a line or lines of body text. It will also position two gadgets at the lower-left and lower-right of the requester. You must also supply the text for both of these gadgets. All text must be in the INTUITEXT format and the strings containing the actual text are NULL terminated. Once the requester appears everything waits until you click on one of those two gadgets. Clicking on the left gadget puts a 1 in register d0 and clicking on the right gadget puts a 0 in d0.

In case you've missed some previous articles I'll review the 20-byte INTUITEXT structure. The first part contains:

BYTE	
0	the front pen color (foreground)
1	the back pen color (background)
2	drawmode (JAM1, JAM2, XOR,...)
3	padding
4	offset in pixels from the left edge
6	offset in pixels from the top edge
8	font pointer, if any
12	pointer to the NULL-terminated text string
16	pointer to the next INTUITEXT structure, if any

followed by the text in this format -

BODYSTRING DC.B 'OK to ZOOM?',0

The requesters in this program will ask if you want to zoom and if you want a small or large picture.

I mentioned that you call up the requester using the Intuition function AutoRequest. But once it's there, how does the program know when you click in a gadget, and which one? Again, fortunately, the requester makes its own set of IDCMP flags temporarily replacing the ones you assign to your window. The flags will note which of the two gadgets you select, store a corresponding value in d0, restore your window IDCMP flags, and close the requester (actually inactivate it).

Plain Vanilla

This program will also react to key presses if you include the VANILLAKEY ($200000) IDCMP flag in your window structure. I used a modified INTUIMESSAGE structure that eliminates menus since all I wanted were IDCMP flags, ASCII values, and mouse X/Y coordinates. The INTUIMESSAGE structure is:

BYTE	
0	the IM.MESSAGE
20	IM.CLASS - IDCMP flags
24	IM.CODE - menu/item/subitem, ASCII values
26	IM.QUALIFIER - rawkey codes
28	IM.ADDRESS - this function address
32	IM.MOUSEX - X coordinate
34	IM.MOUSEY - Y coordinate
36	IM.SECONDS - current time in seconds
40	IM.MICROS - time in tenths of seconds
44	IM.IDCMPWINDOW - window address for the IDCMP
48	IM.SPECIALLINK - for system/special use

When a message is received, the program first checks to see which IDCMP flag it is. If it is a MOUSEBUTTON, the program will go to the zoom routine. If it is a MOUSEMOVE, the current coordinates of the points are stored in MOUSEX and MOUSEY. If the IDCMP flag is VANILLAKEY then you pressed a key and the ASCII value is checked in IM.CODE. The program reacts to the following keys:

s - start drawing
c - go to the coefficient screen
s - switch/toggle the picture size
0 through 9 - change palettes
q - quit the program

There are 10 palette values at the end of the program and any time you press 0-9 on the keyboard or keypad the program will use the LoadRGB function to immediately change to that palette.

Mr. Newton

Now that we know how the program does things, what does it do? Well, it solves simple equations up to degree 7 and graphically shows you the answers. To solve for roots, the program uses the Newton method, which says that if you think the answer (root) to an equation is X, then a better answer is X-F(X)/F'(X). F(X) is simply the equation you're trying to solve. F'(X) is the first derivative of the equation. Perhaps you remember from high school that the first derivative of A*X^N is N*A*X^(N-1); the first derivative of 2X^3 is 6X^2.

Let's try an example and solve X^2=16 or X^2-16=0. The formula says that our first guess of X can be replaced by X-(X^2-16)/2X or (X^2+16)/2X. Let's try 2 as the first guess. Putting in 2 for X results in 20/4 or 5. Now use 5 in the formula to get 41/10 or 4.1. Try 4.1 and get 32.81/8.2 or 4.0012. Getting pretty close aren't we? If we establish a tolerance factor, the difference between two successive guesses will approach and finally reach that tolerance. The number of iterations it takes to reach that tolerance gets converted to a color palette value and the initial guess is PSET on the screen in that color value.

Complex Numbers

But just using different values for X will only give us a nice colored line—not really worth the effort. We need something that will also use the up/down part of the screen, the Y-axis. In previous articles about the Mandelbrot and Julia sets I discussed complex numbers, those numbers comprised of a real and imaginary part using "i" (the square root of -1). The Newton formula works just as well for complex numbers Z; remember that Z=X+iY.

Again, using Z^2-16=0, our formula is (Z^2+16)/2Z. But now we need to separate Z into its real and imaginary terms. Since Z^2=X^2-Y^2+2XYi we can rewrite this as ((X^2-Y^2+16)+2i(XY))/(2X+2iY). This is a complex number divided by a complex number. In simple terms, (A+Bi)/(C+Di)=(AC+BD)/(C^2+D^2)+i(BC-AD)/(C^2+D^2). Now let's look at a 7th degree equation using C7 to C0 as the coefficients. Combining the terms we get

```
6*C7*Z^7+5*C6*Z^6+4*C5*Z^5+3*C4*Z^4+2*C3*Z^3+C2*Z^2-C0
7*C7*Z^6+6*C6*Z^5+5*C5*Z^4+4*C4*Z^3+3*C3*Z^2+2*C2*Z+C1)
```

There is an easy way to compute the real and imaginary parts of Z^7, Z^6, etc. Starting with an initial guess of A+Bi just keep following this repetition:

```
REAL TERMS                                        IMAGINARY TERMS
RT1=A                                                  IT1=B
RT2=RT1*RT1-IT1*IT1                               IT2=2*RT1*IT1
RT3=RT1*RT2-IT1*IT2                               IT3=RT1*IT2+RT2*IT1
RT4=RT2*RT2-IT2*IT2                               IT4=2*RT2*IT2
RT5=RT2*RT3-IT2*IT3                               IT5=RT2*IT3+RT3*IT2
RT6=RT3*RT3-IT3*IT3                               IT6=2*RT3*IT3
RT7=RT3*RT4-IT3*IT4                               IT7=RT3*IT4+RT4*IT3
```

These types of repetitive formulas are very easy to handle with macros; just assign RT1=A and IT1=B. Notice that all the real terms are the difference between multiplied terms. You would pass a location and four terms to the real term macro. It will multiply the first two terms, save the product, multiply the next two terms, subtract from the product and store the result in the passed location. All the even imaginary terms are twice the product of two terms, and all the odd imaginary terms are the sum of multiplied terms just as in the real term macro, except that the products are added together.

Let's take a look again at that equation above that solves everything. The first term in the numerator is 6*C7*Z^7. Express this as 6*C7*RT7 and 6*C7*IT7; let A1 equal the real part and B1 equal the imaginary part. The next real part, 5*C6*RT6, gets added to A1 and the next imaginary part, 5*C6*IT6, is added to B1. At the end of the numerator C0 is subtracted from A1. In the denominator let A2 equal 7*C7*RT6 and let B2 equal 7*C7*IT6. Keep adding the real terms to A2 and the imaginary terms to B2 and at the end of the denominator add C1 to A2.

Now you have a new complex number (A1+iB1)/(A2+iB2). Multiply the top and bottom by (A2-iB2) to get a new real number (A1*A2+B1*B2) / (A2*A2+B2*B2) and a new imaginary number i(A2*B1-A1*B2) / (A2*A2+B2*B2). Before you actually divide though, be sure to check that the denominator isn't 0; it's not nice to divide by 0 and the computer will strongly object. Call the new real number AA and compare it to the starting A. If it's within your previously defined tolerance, compare the new imaginary number BB to the original B. If it's also within your tolerance, then this new number solves the equation and is a root. PSET the ACROSS/DOWN location with a color based on how many iterations it took to find the root. If either new value is not within the tolerance, you don't have a root, so replace A and B with AA and BB and start all over again.

The program allows for up to 47 iterations. If you haven't reached a root within this number of iterations, that point is either bouncing between two roots or heading to plus or minus infinity. Each iteration rounds off numbers, that's why I used double-precision for the computations.

The Program

When the program starts, you are presented with eight strings showing the current values of coefficients C7 through C0 and four strings showing the left/right and top/bottom boundaries of the area you'll be plotting. The strings were drawn using macros I developed in an earlier article (V3.2). If you haven't seen this article either, you really should consider subscribing. The string macros are in MENU.I included on this disk.

Change any string value by clicking in its box; replace values or use RIGHT AMIGA/X to clear it and type in new values. Press <enter> after each changed string. When you're ready to draw, press "s"; a requester will appear asking if you want a small picture (128 x 128) or a large one (320 x 200). After you select a size, the picture will start drawing. While it's drawing, you can press any of the values "0" through "9" to change the palette, or press "x" to start drawing the other size picture. I usually only change size from small to large. Also, at any time you can position the cursor at the upper-left corner of a zoom area, press the LMB and drag down to the lower-right corner. When you release the LMB, a requester will ask if it's O.K. to zoom. Click in NO, and the zoom box will disappear while the picture continues drawing. Click YES, and a new requester appears asking if you want the SMALL or LARGE size; click on the one you want and the new picture starts drawing. At any time press "c" to return to the coefficient menu or press "q" to quit the program.

The Amiga 3000 has some features that are rather cryptic; for example, the first line in my S:user-startup is CPU FASTROM BURST. This seems to be a requirement for most of the programs I run; very few programs require me to REM this line. If you have any problems running this program (and it's best run from RAM:) ,increase the stack to 30,000 using STACK 30000. This additional memory may be necessary to keep all the strings and requesters from bumping into each other. By the way, you usually have to cold-boot when you change your CPU command.

The Listing

Now let's take a look at Listing1 to see how I programmed all of this. Since I've already talked about many of the routines and the code is pretty heavily documented, I won't spend too much time on the program. Because you can toggle between picture size, the following variables need to go in several routines.

	SMALL	LARGE	
xlength	128	320	for right/left scaling
xlength1	129	320	stop drawing across here
ylength	128	200	for top/bottom scaling
ylength1	129	200	stop drawing down here
xoffset	96	0	to center/correct the display
yoffset	164	199	to center/correct the display.
normalizex	96	0	convert zoom box to coordinates
normalizey	36	0	convert zoom box to coordinates
redraw	0	1	toggle the picture size

The PSET macro uses the offsets to center the small picture and draw the large one correctly. The next four macros compute the real and imaginary terms and their coefficients.

The contents of each string buffer are converted to double-precision values and stored in their proper locations. Note that Ybottom is saved twice since this value changes while the picture is drawing. The variable ALLDONE is necessary since different routines are required when the picture has finished drawing and these are not the same as those used while it's drawing. REDRAW is used to toggle the drawing; its value is XORd with #1 to determine the current picture size.

Next, all the real and imaginary terms are computed and combined to get A1, B1, A2, and B2. The common denominator DE is computed and checked to see if it's 0. Then the new complex numbers AA and BB are computed. Both are compared to the TOLERANCE value ($3EC00000,0); you can change this value located near the end of the program. If AA and BB are both within this tolerance, the current count is ANDed with 31 and used as the color value to PSET the across/down location. If not, the count is increased until it gets to 48 and the program goes to the next point.

After increasing the Xdistance by the Xscale, a check is made for any messages. Key presses go to their corresponding routines and using the LMB will cause a branch to the zoom routine. ZOOM uses the MouseX/Y coordinates to draw a box in the complimentary mode. When you release the LMB the first requester appears and if you want to go ahead it's followed by the second requester asking for the picture size. Depending on which size you pick the proper variable sizes are filled.

NEWCOORDINATES uses these variables to convert those MouseX/Y locations to the proper grid coordinates and then computes new start and end locations for your display. At the completion of drawing, the same keyboard options are still available. ALLDONE is cleared so that routines will branch back to NOW_WHAT instead of trying to continue drawing. Note the IDCMP and window flags used in MYWINDOW to get the proper IDCMP results.

There are 10 palettes at the end of the program. Feel free to modify them in any way. If you want the coefficients or display area to initially appear differently when the coefficient menu appears change their gadget buffers. And you can modify the requesters by changing their text strings. The very end of the listing contains some sample equations and their display area that you might want to try. If you make any changes to this program assemble it using A68K as NEWTON.ASM and BLINK it as NEWTON.O. Run the program as NEWTON. This program, A68K, BLINK, and all the required "include" files are on the magazine disk. I've also included on the disk a picture made from enlarging a portion of the equation Z^7-1 using palette 2.

Don't Forget

When you assemble or run this program, remember to have cold-booted with the line CPU FASTROM BURST in the s:user-startup script. And use the command STACK 30000 if you have problems running NEWTON. Some equations may go to infinity very quickly, especially if you're using large coefficients; this may cause an overflow and crash the program. I've not included a check for maximum values since that would slow things down but you might want to add one. And finally, while running the program you can use the following keys:

s - start drawing
c - return to coefficient menu
x - toggle picture size
0 to 9 - change palettes
q - quit the program

TABLE II
AUTOREQUEST (Intuition offset -348)

REGISTER	
a0	window - pointer to your window
a1	bodytext - pointer to intuitext explaining your requester
a2	positivetext - pointer to left gadget intuitext
a3	negativetext - pointer to right gadget intuitext
d0	positiveflags - IDCMP flags for left gadget, usually 0
d1	negativeflags - IDCMP flags for right gadget, usually 0
d2	width - requester width in pixels
d3	height - requester height in pixels

(returns d0=1 if left gadget selected, d0=0 if right gadget selected)

```
;listing1
;Newtons method for root solving in double precision
   idnt   [illegible]
  include 'exemacros.i'
  include 'intmacros.i'
  include 'dosmacros.i'
  include 'gfxmacros.i'
  include 'dpmathmacros.i'
  include 'menu.i'

count equr d3
across  equr d4
down    equr d5
depth equ 5

xpset  macro          ;across,down-
  movea.l rplot,a1
   move.w  \1,d0
   add.l   xoffset,d0          ;adjust across
   move.l  yoffset,d1          ;adjust down
   sub.w   \2,d1
   ext.l   d0
   ext.l   d1
  move.l gfxbase(pc),a6
   jsr      -324(a6)
   endm

realtermmacro ;loc,term1,term2,term3,term4
   movedp  \4,d0 ;term1
   movedp  \5,d2 ;term4
   muldp         ;term1 * term4
   movedp  d0,d6
   movedp  \2,d0 ;term1
   movedp  \3,d2 ;term2
   muldp         ;term1 * term2
   movedp  d6,d2 ;term1 * term4
   subdp         ;(term1 * term2) - (term1 * term4)
   movedp  \1    ;save
   endm
imagterm macro ;loc,term1,term2
   movedp  \2,d0
   movedp  \3,d2
   muldp          ;term1 * term2
   movedp  d0,d2
   adddp          ;2 * term1 * term2
   movedp   \1
   endm
imagterm2 macro ;loc,term1,term2,term3,term4
   movedp  \2,d0 ;term1
   movedp  \3,d2 ;term2
   muldp         ;term1 * term2
   movedp  d0,d6 ;save this
   movedp  \4,d0 ;term3
   movedp  \5,d2 ;term4
   muldp         ;term3 * term4
   movedp  d6,d2 ;term1 * term2
   adddp         ;(term1 * term2) + (term3 * term4)
   movedp  \1    ;save it
   endm
mulcoef macro ;loc,term1,term2(,term3)
   movedp  \2,d0 ;term1
   movedp  \3,d2 ;term2
   muldp         ;term1 * term2
   ifeq    NARG-4 ;4 variables?
   movedp  \4,d2 ;term3
   muldp         ;term1 * term2 * term3
   endc
```

```
   movedp   \4,d2 ;location
   adddp          ;add location
   movedp   \1    ;save it
   endm

getit    macro    ;branch to \1 if no msg
  movea.l windowptr,a0
  movea.l wd_UserPort(a0),a0
   syslib  getmsg
   tst.l    d0
   beq      \1
   movea.l  d0,a1
   move.l  im_Class(a1),d2  ;IDCMP
   move.w  im_Code(a1),d3   ;ASCII
  move.w  im_MouseX(a1),d4 ;Xcoordinate
  move.w  im_MouseY(a1),d5 ;Ycoordinate
   syslib  replymsg
   endm

start
   move.l sp,stack               ;save stack pointer

open_libs                        ;open all the libraries
we_need
   openint    int,done
   opendos    dos,close_int
   opengfx    gfx,close_dos
   openlib    dpmath,close_gfx
   doslib      output
   move.l      d0,conhandle

settings
   cls
   linefeed
   right    15
   oldstack
   print   titlemsg
   normal
   linefeed
   print    amsg
   linefeed
   print    bmsg
   linefeed
   print    pglmsg
   linefeed
   print    cmsg
   linefeed
   print    zoommsg
   linefeed
   linefeed
   right    45
  style   italics,orange,black ;not 1000 colors
   print    lmsg
   normal
   linefeed
loop
   bra      loop
setup:             ;open a screen of 320 x 200
open_screen
  openscreen myscreen,close_libs
open_window
  openwindow mywindow,close_screen
   menu        [illegible]
  palette    colormap(pc),32
msg_check
   getit    msg_check
   beq.s    msg_check
  cmpi.w  #'-',d1 ;start?
  beq.s   dp_newton_demo
  bra.s   mag_check
dp_newton_demo
[illegible]
  removegadgets
  lea     gadget1buffer(pc),a0
  bsr     convertdp
  movedp  c7

  lea     gadget2buffer(pc),a0
  bsr     convertdp
  movedp  c6

  lea     gadget3buffer(pc),a0
  bsr     convertdp
  movedp  c5

  lea     gadget4buffer(pc),a0
  bsr     convertdp
  movedp  c4

  lea     gadget5buffer(pc),a0
  bsr     convertdp
  movedp  c3

  lea     gadget6buffer(pc),a0
  bsr     convertdp
  movedp  c2

  lea     gadget7buffer(pc),a0
  bsr     convertdp
  movedp  c1

  lea     gadget8buffer(pc),a0
  bsr     convertdp
  movedp  c0

  lea     gadget9buffer(pc),a0
  bsr     convertdp
  movedp  xleft

  lea     gadget10buffer(pc),a0
  bsr     convertdp
  movedp  ytop

  lea     gadget11buffer(pc),a0
  bsr     convertdp
  movedp  xright

  lea     gadget12buffer(pc),a0
  bsr     convertdp
  movedp  ybottom
  movedp  saveybottom
  bsr     sequence2  ;draw SMALL or LARGE
[illegible]
  move.l  xlength,d0
  ltodp
  movedp  d0,d6
  movedp  xright,d0
  movedp  xleft,d2
  subdp
  movedp  d6,d2
  divdp
  movedp  xinc             ;x scale

  movedp  saveybottom,ybottom
```

```
  move.l  ylength,d0
  ltodp
  movedp  d0,d6
  movedp  ytop,d0
  movedp  ybottom,d2
  subdp
  movedp  d6,d2
  divdp
  movedp  yinc            ;y scale
  bra     newwin

convertdp
  moveq.l #0,d0
  moveq.l #0,d1
  moveq.l #0,d4           ;sign positive
  moveq.l #0,d5
  moveq.l #0,d7           ;# characters input
decimal
  sub.l   d2,d2           ;clear d2: decimal flag
negative
  cmpi.b  #'-',(a0)       ;a minus sign?
  bne.s   positive
  bset    #31,d4          ;if so set last bit in d4
  addq.l  #1,a0           ;move over one space
positive
getdigit
  move.b  (a0)+,d5        ;next ascii in d5
  cmpi.b  #'.',d5         ;a decimal?
  bne.s   testdigit
  move.w  #1,d2           ;if so, flag it
  clr.l   d7              ; and clear d7
  bra.s   getdigit
testdigit
  cmpi.b  #'9',d5         ;above '9' ?
  bhi.s   zero_check      ;branch if so
  cmpi.b  #'0',d5         ;below '0' ?
  blt.s   zero_check      ;branch if so
  andi.l  #$0F,d5         ;convert ASCII to decimal

  movem   d0,d2           ;must multiply d0 * 10
  asl.l   #1,d1           ;d0 * 2
  roxl.l  #1,d0
  asl.l   #1,d3           ;d2 * 8
  roxl.l  #1,d2
  asl.l   #1,d3
  roxl.l  #1,d2
  asl.l   #1,d3
  roxl.l  #1,d2
  moveq.l #0,d6
  add.l   d3,d1           ;d0 = d0 + d2
  addx.l  d2,d0
  add.l   d5,d1           ; + this digit
  addx.l  d6,d0

  addq.w  #1,d7           ;number of characters done
  cmpi.w  #16,d7          ;up to 16 ?
  bne     getdigit
[illegible]
  ;moveq  #1,d1           ;optional error flag
  ;rts
  bra     close_window    ;exit - close everything
zero_check
  move.l  a0,a5           ;return string location
  tst.l   d1              ;don't try to convert if
  bne.s   not_exponent    ;branch if not 0
  tst.l   d0              ;check second half
  beq.s   dp_done         ;branch if value is 0
```

The product offering continues

rufos

The API for discriminating programmers. ($50)

rokroot image maker

The simple and effective drawing/animation program. ($30)

ruf

The .h assembly utility. ($15)

Call for details.

Send disk for demo disk in return.

ROKROOT software
5411 37th Avenue South
Seattle, WA 98118-2212
206-722-6258 fax: 206-722-0664
American Express welcome.

AC'S GUIDE SUMMER 1993

Looking for a specific product for your Amiga but you don't know who makes it? Want a complete listing of all the Fred Fish software available? Just looking for a handy reference guide that's packed with all the best Amiga software and hardware on the market today?

If so, you need AC's *GUIDE for the Commodore Amiga*. Each *GUIDE* is filled with the latest up-to-date- information on products and services available for the Amiga. It lists public domain software, user's groups, vendors, and dealers. You won't find anything like it on the planet; and you can get it only from the people who bring you the best monthly resource for the Amiga, *Amazing Computing*.

So to get all this wonderful information, call **1-800-345-3360** today and talk to a friendly Customer Service Representitive about getting your *GUIDE*. Or stop by your local dealer and demand your copy of the latest *AC's GUIDE for the Commodore Amiga*.

List of Advertisers

Company	Page Number
Armadillo Computing	15
AMOS	CII
Delphi Noetic Systems	7
Devine Computers	13
Digital Creations	CIV
Dineen Edwards	CIII
Memory Management	18
Oregon Research	6
Rokroot Software	48

WHAT'S ON IT?

AC's TECH 3.4 Disk Includes Source & Executables For:

- F-BASIC 5.0 review
- Quick Menus for True BASIC Programs
- Time Efficient Animations
- True BASIC Input Mask
- Programming the Amiga in Assmely Language
- Have Your Own Custom 3-D Graphics Package Part II

AND MORE!

AC's TECH 3.4 CHECK IT OUT!

AC's TECH Disk

Volume 3, Number 4

A few notes before you dive into the disk!

- You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

- In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' which is provided in the C: directory. lharc archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh*, type *lharc x foo*
  For help with lharc, type *lharc ?*
  Also, files with 'lock' icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.

We pride ourselves in the quality of our print and magnetic media publications. However, in the highly unlikely event of a faulty or damaged disk, please return the disk to *PiM Publications, Inc.* for a free replacement. Please return the disk to:

AC's TECH
Disk Replacement
P.O. Box 2140
Fall River, MA 02720-2140

Be Sure to Make a Backup!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc. their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright PiM Publications, Inc., and may not be duplicated in any way. The purchaser, however, is encouraged to make an archival backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work twice to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PiM Publications or any of its agents is not responsible for any damages incurred while attempting this project.

```
get_exponent
   move.l   #$43f,d6            ;maximum exponent
l7
   subq.l   #1,d6               ;decrease exponent
   asl.l    #1,d1               ;m1 * 2
   roxl.l   #1,d0               ;rotate with carry from d1
   bcc.s    l6                  ;branch if no carry
   moveq.l  #11,d5              ;move right 12 bits

shift_right
   lsr.l    #1,d0
   roxr.l   #1,d1
  dbra.s  d5,shift_right
adjust_exponent
   swap     d6                  ;move to left word
   asl.l    #4,d6               ;move to left and
   or.l     d6,d0               ;put in d0
fraction_check
   cmpa.l   #0,a2               ;any decimal
   beq.s    do_sign             ;no
   subq.l   #1,d7               ;decrease number of digits
  bmi.s    do_sign             ;no digits after the decimal

decimal_adjust
  move.l   #$40240000,d2       ;dp 10
   moveq.l  #0,d3              ; part 2
   divdp
  dbra.s  d7,decimal_adjust ;do for all digits

do_sign
   or.l     d4,d0               ;add sign to d0
do_done
   moveq.w  #0,d6               ;all is 'ok'
   rts

showit
  movem.l d0-a6,-(sp)
   pcia
  move.l   #1,alldone          ;completed? flag
  move.w   #0,down
l1
  movedp   xleft,xx
  move.w   #0,across
l2
  movedp   xx,a
  movedp   ybottom,b
  move.w   #0,count            ;clear the count
l3
  movedp   a,rt1
  movedp   b,it1

  realterm   rt2,a,a,b,b
  imagterm   it2,a,b

  realterm   rt3,a,rt2,b,it2
  imagterm   it3,a,it2,rt2,b

  realterm   rt4,rt2,rt2,it2,it2
  imagterm   it4,rt2,it2

  realterm   rt5,rt2,rt3,it2,it3
  imagtermo  it5,rt2,it3,rt3,it2

  realterm   rt6,rt3,rt3,it3,it3
  imagterm   it6,rt3,it3

  realterm   rt7,rt3,rt4,it3,it4
```

```
  imagtermo  it7,rt3,it4,rt4,it3

  moveq    #0,d0 ;clear terms
   moveq    #0,d1
   movedp   a1
   movedp   b1
   movedp   a2
   movedp   b2

  getcoef a1,sixdp,c7,rt7
  getcoef b1,sixdp,c7,it7
  getcoef a2,sevendp,c7,rt6
  getcoef b2,sevendp,c7,it6

  getcoef a1,fivedp,c6,rt6
  getcoef b1,fivedp,c6,it6
  getcoef a2,sixdp,c6,rt5
  getcoef b2,sixdp,c6,it5

  getcoef a1,fourdp,c5,rt5
  getcoef b1,fourdp,c5,it5
  getcoef a2,fivedp,c5,rt4
  getcoef b2,fivedp,c5,it4

  getcoef a1,threedp,c4,rt4
  getcoef b1,threedp,c4,it4
  getcoef a2,fourdp,c4,rt3
  getcoef b2,fourdp,c4,it3

  getcoef a1,twodp,c3,rt3
  getcoef b1,twodp,c3,it3
  getcoef a2,threedp,c3,rt2
  getcoef b2,threedp,c3,it2

   getcoef a1,c2,rt2
   getcoef b1,c2,it2
  getcoef a2,twodp,c2,rt1
  getcoef b2,twodp,c2,it1

   subdp    a1,c0
   movedp   a1
   adddp    a2,c1
   movedp   a2

getde
   movedp   a2,d0
   movedp   d0,d2
   muldp              ;a2 * a2
   movedp   d0,d6
   movedp   b2,d0
   movedp   d0,d2
   muldp              ;b2 * b2
   movedp   d6,d2 ;a2 * a2
   adddp           ;(a2 * a2) + (b2 * b2)
   tst.l    d0
   beq      l1      ;is 0
   movedp   de     ;save denominator

getaa
  imagtermo  aa,a1,a2,b1,b2 ;(a1 * a2) + (b1 * b2)
  movedp   de,d2
  divdp           ;((a1 * a2) + (b1 * b2)) / de
  movedp   aa    ;save it
getbb
 realterm bb,a2,b1,a1,b2 ;(a2 * b1) - (a1 * b2)
  movedp   de,d2
  divdp              ;((a2 * b1) - (a1 * b2)) / de
  movedp  bb     ;save ()

compare
  movedp  a,d6
  movedp  aa,d7
  subdp          ;b - ba
  absdp          ;abs(b - ba)
  movedp  tolerance,d2
  cmpdp          ;d0=1 if greater
                 ;d0=0 if equal
                 ;d0=-1 if less
  bpl.s   continue

  movedp  b,d6
  movedp  bb,d7
  subdp          ;b - bb
  absdp          ;abs(b - bb)
  movedp  tolerance,d2
  cmpdp
  bpl.s   continue

  move.w  count,d0    ;count = color
  andi.w  #31,d0
  foreground          ;set pen
  mpset   (pointdown) ;plot the point
  bra.s   l4
continue
  movedp  aa,a
  movedp  bb,b
  addq.w  #1,count     ;increase count
  cmp.w   #32,count    ;up to maximum yet ?
  bne     l3           ;branch if not
l4
  move.l  xx(pc),d6
  move.l  xx+4(pc),d7
  move.l  xinc(pc),d2
  move.l  xinc+4(pc),d3
  addp
  movedp  xx

check_for_message
  @pin    no_message
  cmpi.l  #mousebuttons,d2
  beq     zoom
  tst.l   d1
  beq     no_message
  andi.l  #$000000ff,d1 ;ASCII value
  cmpi.w  #'1',d1     ;#1
  beq     do_palette1
  cmpi.w  #'2',d1     ;#2
  beq     do_palette2
  cmpi.w  #'3',d1     ;#3
  beq     do_palette3
  cmpi.w  #'4',d1     ;#4
  beq     do_palette4
  cmpi.w  #'5',d1     ;#5
  beq     do_palette5
  cmpi.w  #'6',d1     ;#6
  beq     do_palette6
  cmpi.w  #'7',d1     ;#7
  beq     do_palette7
  cmpi.w  #'8',d1     ;#8
  beq     do_palette8
  cmpi.w  #'9',d1     ;#9
  beq     do_palette9
  cmpi.w  #'0',d1     ;#0
  beq     do_palette10
```

```
  cmpi.w  #'r',d1
  beq.s   new_values
  cmpi.w  #'q',d1
  beq     close_window
  cmpi.w  #'s',d1
  beq.s   redraw_it
  bra     no_message
redraw_it
  tst.l   #1,redraw
  beq     draw_small
  bra     draw_large
new_values
  print
  closewindow
  bra     make_window
do_palette1
  palette colormap1(pc),d2
  bra     no_message
do_palette2
  palette colormap2(pc),d2
  bra     no_message
do_palette3
  palette colormap3(pc),d2
  bra     no_message
do_palette4
  palette colormap4(pc),d2
  bra     no_message
do_palette5
  palette colormap5(pc),d2
  bra     no_message
do_palette6
  palette colormap6(pc),d2
  bra     no_message
do_palette7
  palette colormap7(pc),d2
  bra     no_message
do_palette8
  palette colormap8(pc),d2
  bra     no_message
do_palette9
  palette colormap9(pc),d2
  bra     no_message
do_palette10
  palette colormap10(pc),d2
  bra     no_message

zoom
  andi.l  #$0000ffff,d5 ;mouse x
  andi.l  #$0000ffff,d6 ;mouse y
  move.l  d5,startx
  move.l  d6,starty
  mode    complement
lmb_down
  @pfm    lmb1
  cmpi.w  #selectup,d1
  beq     lmb_up
lmb1
  andi.l  #$0000ffff,d5 ;mouse x
  andi.l  #$0000ffff,d6 ;mouse y
  move.l  d5,endx
  move.l  d6,endy
  box     startx,starty,endx,endy,d3
  delay   1
  box     startx,starty,endx,endy,d3
  bra     lmb_down
lmb_up
  box     startx,starty,endx,endy,d3
  mode    jam1
requester1
  move.l  windowptr,a0
  lea     bodytext,a1
  lea     positivetext,a2
  lea     negativetext,a3
  moveq   #0,d0
  moveq   #0,d1
  move.l  #170,d2
  move.l  #50,d3
  intlib  autorequest
  tst.l   d0
  bne     new_coordinates
  mode    complement
  box     startx,starty,endx,endy,21
  mode    jam1
  tst.l   alldone
  beq     now_what ;finished drawing
  bra     check_for_message
new_coordinates
check_them
  move.l  startx,d0
  move.l  endx,d1
  cmp.l   d1,d0
  blo.s   nc1   ;startx < endx
  exg     d1,d0
  move.l  d0,startx
  move.l  d1,endx
nc1
  move.l  starty,d0
  move.l  endy,d1
  cmp.l   d1,d0
  blo.s   nc2   ;starty < endy
  exg     d0,d1
  move.l  d0,starty
  move.l  d1,endy

nc2
  move.l  startx,d0
  sub.l   normalizex,d0
  fltdp
  movedp  xinc,d2
  muldp
  adddp   xleft
  movedp  newxleft
  move.l  endx,d0
  sub.l   normalizex,d0
  fltdp
  movedp  xinc,d2
  muldp
  adddp   xleft
  movedp  xright
  movedp  newxleft,xleft

  move.l  starty,d0
  sub.l   normalizey,d0
  fltdp
  movedp  yinc,d2
  muldp
  movedp  d0,d2
  movedp  ytop,d0
  subdp
  movedp  newytop
  move.l  endy,d0
  sub.l   normalizey,d0
  fltdp
  movedp  yinc,d2
```

```
  muldp
  movedp  d0,d2
  movedp  ytop,d0
  subdp
  movedp  ybottom
  movedp  [illegible]
  movedp  newytop,ytop

requester2
  move.l  windowptr,a0
  lea     bodytext2,a1
  lea     smalltext,a2
  lea     largetext,a3
  moveq   #0,d0
  moveq   #0,d1
  move.l  #170,d2
  move.l  #50,d3
  intlib  autorequest
  tst.l   d0
  beq     draw_large
draw_small
  move.l  #128,xlength
  move.l  #128,xlength1
  move.l  #128,ylength
  move.l  #128,ylength1
  move.l  #96,xoffset
  move.l  #164,yoffset
  move.l  #96,normalizex
  move.l  #44,normalizey
  move.l  #0,redraw
  bra     scale
draw_large
  move.l  #320,xlength
  move.l  #320,xlength1
  move.l  #200,ylength
  move.l  #200,ylength1
  move.l  #0,xoffset
  move.l  #189,yoffset
  move.l  #0,normalizex
  move.l  #0,normalizey
  move.l  #1,redraw
  bra     scale
no_message
  addq.w  #1,across            ;across the space
  cmp.l   xlength1,across      ;all way across yet ?
  bne     l1                   ;branch if not

  adddp   ybottom,yinc         ;increment y to yscale
  movedp  ybottom
  addq.w  #1,down              ;down the space
  cmp.l   ylength1,down        ;all way down yet ?
  bne     l1                   ;branch if not
  move.l  #0,alldone           ;finished drawing

now_what
  bsr     now_what_done
  cmp.l   #mousebuttons,d2
  beq     [illegible]
  and.l   #$0000ff0,d1
  cmpi.w  #'1',d1    ;#1
  beq.s   do_palette1
  cmpi.w  #'2',d1    ;#2
  beq     do_palette2
  cmpi.w  #'3',d1    ;#3
  beq     do_palette3
  cmpi.w  #'4',d1    ;#4
  beq     do_palette4
```

```
    cmpi.w  #'5',d3     ;#5
    beq     do_palette5
    cmpi.w  #'6',d3     ;#6
    beq     do_palette6
    cmpi.w  #'7',d3     ;#7
    beq     do_palette7
    cmpi.w  #'8',d3     ;#8
    beq     do_palette8
    cmpi.w  #'9',d3     ;#9
    beq     do_palette9
    cmpi.w  #'0',d3     ;#0
    beq     do_palette0
    cmpi.w  #'n',d3
    beq     new_values
    cmpi.w  #'c',d3
    beq     redraw_it
    cmpi.w  #'q',d3
    beq     close_window
no_what_then
    bra     new_what

do_palette1
  palette colormap1(pc),32
    bra     new_what
do_palette2
  palette colormap2(pc),32
    bra     new_what
do_palette3
  palette colormap3(pc),32
    bra     new_what
do_palette4
  palette colormap4(pc),32
    bra     new_what
do_palette5
  palette colormap5(pc),32
    bra     new_what
do_palette6
  palette colormap6(pc),32
    bra     new_what
do_palette7
  palette colormap7(pc),32
    bra     new_what
do_palette8
  palette colormap8(pc),32
    bra     new_what
do_palette9
  palette colormap9(pc),32
    bra     new_what
do_palette0
  palette colormap0(pc),32
    bra     new_what
close_window
  closewindow
close_screen
  closescreen
close_lib
  closelib dpmath
close_gfx
  closelib gfx
close_dos
  closelib dos
close_int
  closelib int
done
  move.l  stack,sp
  rts
```

```
  even

stack      dc.l  0          ;possible library locations
gfxbase    dc.l  0
intbase    dc.l  0
dosbase    dc.l  0
dpmathbase dc.l  0
                            ;library names
gfx        dc.b  'graphics.library',0
  even
int        dc.b  'intuition.library',0
  even
dos        dc.b  'dos.library',0
  even
dpmath     dc.b  'mathieeedoubbas.library',0
  even

myscreen:
  dc.w 0,0,320,200,depth    ;depth 1-5
  dc.b 0,1
  dc.w 0
  dc.w customscreen
  dc.l 0,0,0,0
  even
mywindow
  dc.w 0,0,320,200
  dc.b 0,1
 dc.l mousebuttons|mousemove|vanillakey
 dc.l borderless|activate|reportmouse
  dc.l gadget1,0
  dc.l 0
  dc.l 0,0
  dc.w 0,0,0,0
  dc.w customscreen

  even
xleft   dc.l  0,0           ;reserve space for both
parts
xright  dc.l  0,0
xx      dc.l  0,0
xinc    dc.l  0,0
ybottom dc.l  0,0
ytop    dc.l  0,0
yinc    dc.l  0,0
startx  dc.l  0
starty  dc.l  0
endx    dc.l  0
endy    dc.l  0
xlength dc.l  128
```

The complete Source Code and Executables can be found on the AC's TECH Disk.

Please write to:
William P. Nee
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722

Re Color

by Dave Senger

Last December, I upgraded my Amiga 2000 from Operating System 1.3.2 to O.S. 2.1. I love it. My new O.S. is slick and full of conveniences, and the new Workbench screen looks sharp, with its business-like colors and its patterned windows. Thoughtful touches, such as the Screen Blanker, and the very well designed Mouse Accelerator (at last!), tell me that Commodore is working hard and paying attention to details.

Along with all the improvements came an annoyance, however. Most of the icons on my old floppies—and I have a lot of old floppies—looked terrible on the new Workbench screen. The change in colors from blue, white, black, and orange on the old screen, to grey, black, white, and light blue on the new one, often makes icons designed for the old screen look like candidates for the morgue. The reversal of white and black to black and white is especially unhelpful.

Commodore is aware of the problem, and has provided a solution—the Recolor function on the Extras menu of IconEdit, in the Tools drawer. Using this function to recolor your old icons is very simple. Just drag an icon into IconEdit's large window, then select Recolor from the menu, or use the <Right Amiga>-M key combination. In two or three seconds the icon will be recolored, and all you will have left to do is save it.

Recoloring half-a-dozen icons this way works well enough, but recoloring a thousand could get tedious. IconEdit makes no provision for batch processing. Commodore has made ARexx a standard feature of the new operating systems, so it would seem reasonable for Commodore utilities to come equipped with ARexx interfaces, but, as far as I know, Ed is the only one that does. The absence of ARexx interfaces in at least a few of these utilities, such as IconEdit, can't be an oversight. The need is too obvious. The Commodore people have been turning out so much hardware and software lately that they probably just never got around to it.

I couldn't find a way to use IconEdit in conjunction with either an AmigaDOS or ARexx script to batch-process icons, so I wrote my own utility in ARexx, which is ideal for the job. You could write this utility in any number of languages, but only two—ARexx, and the AmigaDOS script language—are available to everyone with one of the new operating systems, who are the only users who need to recolor icons. Of the two, ARexx, with its powerful string- and file-handling functions, is by far the easiest to use.

The utility is called RecolorIcons.rexx, and you will find it listed at the end of this article. It can be used to recolor a single icon at a time, like IconEdit, but I wrote it so that I could recolor many icons in one batch, with a minimum of effort. I've used it to recolor over 2000 icons on dozens of floppies, and so far the script has digested every icon I've fed it, including some that break Commodore's programming rules. There are probably a few icons out there that the script won't handle. I'm sure that I haven't anticipated all the ways in which an ingenious programmer can break the rules. But it should recolor 99% of pre-O.S. 2 icons, and 100% of the legal ones.

ARexx Requirements

For this script to work, ARexx must be set up correctly on your system. Both rexxsyslib.library and rexxsupport.library must be available.

You can start ARexx manually and load libraries one by one, but I use ARexx so much that I want it available all the time. I added the lines below to my Startup-Sequence, so that I can forget about details, and just use the language. The Commodore *ARexx User's Guide*, and the *AmigaDOS User's Guide*, recommend that you edit the User-Startup script in your S directory, and leave the Startup-Sequence alone. Unless you feel completely comfortable editing your Startup-Sequence, this is good advice. Also, it wouldn't hurt to keep working backups of both files. That way, if an editing session goes well off the rails, you can always reboot from another disk and switch to the backup until you figure out what went wrong.

```
Assign REXX: SYS:REXXC
   .
   .
   .
SYS:rexxc/RexxMast >NIL:
Resident REXX:RX pure
Resident REXX:HI pure
Resident REXX:RXSET pure
Resident REXX:TCC pure
Resident REXX:TCO pure
Resident REXX:TE pure
Resident REXX:TS pure
Resident REXX:WaitForPort pure
rx LAL.rexx
```

The first line assigns the REXX: device to the Rexxc directory. I put this line with the rest of my Assigns. The first line of the block loads the ARexx interpreter, so that it will always be instantly available to execute any commands directed to it. The next eight lines make several ARexx programs resident. On a hard disk system, this saves a fraction of a second each time you use one, and on a floppy system it saves a couple of seconds. The last line executes the LAL.rexx (Loads ARexx Libraries) script in my Rexxc directory. I got this script from Merrill Callaway's *The ARexx Cookbook*, which gets a lot of use around here. If you don't have all of the libraries listed, you can edit the script or write one of your own.

```
/* LAL.rexx Loads ARexx Libraries */
/* P. 6-16 of The AREXX COOKBOOK, by Merrill Callaway */

L.1='rexxsupport.library'
/* extended functions (DOS, etc.) */

L.2='rexxarplib.library'
/* locations, windows, gadgets */
```

Breathe new life into your old icons with this handy ARexx utility

```
L.3='rexxmathlib.library'
/* sin, tan, cos, and other math functions */

L.4='rexxutil.library'
/* rexxutils */

DO i=1 TO 4
   IF ~SHOW('L',L.i) THEN CALL ADDLIB(L.i,0,-30,0)
   IF ~SHOW('L',L.i) THEN SAY L.i 'failed to open.'

   END
EXIT 0
```

The rexxsyslib.library is loaded automatically when RexxMast is loaded. I got the last three libraries listed from one of the disks I bought with The *ARexx Cookbook*. If you have them, all these libraries should be in your Libs directory.

Using RecolorIcons.rexx

Type in the script using any text editor, such as Ed or MEmacs, and save it to your Rexx directory. CD a Shell to any directory containing one or more icons that you would like to recolor. Enter 'List #?.info', to see the .info files in the directory. To recolor a single icon, enter, for instance:

```
rx RecolorIcons MyFile.info
```

To recolor several icons, omit the file name, and follow the prompts. You can recolor only the icons in your target directory, or also all of the icons in all of the directories contained in your target directory. If your target directory is a disk's root directory, and you make the second choice, you will recolor every icon on the disk, including the disk icon itself (Disk.info), if it has one.

I prefer to use the script as just described, but you don't have to CD your Shell to the target directory. Instead, you can switch the script to any directory you like. Enter, for instance:

```
rx RecolorIcons DF1:MyDir/
```

The script will switch to your target directory and print its name, then wait for you to choose whether to recolor the icons in that directory only, or also the icons in all child directories. If you omit the trailing slash (/), the script will treat 'MyDir' as a file name, and complain that it lacks a .info suffix.

If you have a single-drive system, set up ARexx as outlined under ARexx Requirements, and copy the script to RAM:. To recolor the icons on a disk in DF0:, enter, for instance:

```
rx RAM:RecolorIcons DF0:
```

ARexx multitasks, so you can run the script on more than one directory, or disk, at a time. If I do this on my Amiga 2000, which has only a standard 68000 microprocessor, I don't save any time since each copy runs slower. If you have one of the newer machines with an '030 or '040, running the script on two or three drives at a time may gain some speed. Otherwise, multitasking this script worked fine when I tried it, with one exception. I have about a dozen floppies, all named AMOS. When I tried to recolor the icons on two identically named disks at the same time, using two different Shells, each CD'd to a different drive, the operating system got confused and ran both scripts on one disk. So as long as the disks have different names, there is no problem.

You can also multitask an application or game while the script is running, as long as it doesn't need to use the same drive that the script is using. You don't have to wait around for it to finish. Just check the drive light once in a while, so that you will know when to swap disks and restart the script.

You have probably recolored at least some of your icons, so you will have directories containing some icons which have been recolored, and others which have not been. What I do in this situation is to make a new directory in the target directory, by selecting the New Drawer item from the Windows menu on the Workbench window. Then I drag all the recolored icons, or all the unrecolored icons, whichever are fewer, into the new drawer. Next I recolor the icons in whichever directory contains unrecolored icons. Then I drag all the icons back from the new drawer to the original, and delete the new drawer. Finally, I reposition any icons that need it, and Snapshot them.

If you have a window full of icons open and you run the script on them, you won't see them change colors as they are processed. You have to force the system to redraw the icons before you will see the results of your work. To see your recolored icons, wait until the script has finished, then either select Update from the Window menu of the Workbench screen, or close the window, then re-open it.

That pretty well covers what you need to know to use the script, and at this point you should be able to throw the magazine in a corner, fire up your system, and go to it. But if you would like to know how it works, read on.

The Nitty-Gritty

The Amiga uses .info files to generate most of the icons you see on your screen. If you save a document called MyDoc with your favorite wordprocessor, it will probably attach an icon to the file by also saving a second file called MyDoc.info. This binary file contains all the information that the system needs to display an icon. Amigas equipped with operating systems from O.S. 2 onwards can also generate default icons for files which don't have their own icons, and these default icons are not stored in .info files. This second variety of icon never needs to be recolored, which puts it outside the scope of this article.

Recoloring icons means processing icon .info files, which you can do only if you understand their design. I couldn't find a clear description of icon .info files in any one source, but by scrounging bits and pieces of information from several books, and by examining a number of icon .info files with the C directory's Type Hex command, I eventually arrived at a pretty clear idea of their layout.

To start with, every icon .info file contains from three to six structures. A structure is like a standardized record. Consistency is the key idea in both cases. In an insurance company's files on policy-holders, for example, the policy number is always found in the same location in each record. The surname always appears in another location. The policy number never appears in the surname's location, or vice-versa.

The Amiga's operating system, and most applications software, are chock-full of structures, and you can do almost no serious programming in C or Assembler without running into them. The operating system contains a number of libraries full of pre-written routines that programmers can use to do hundreds of things such as allocating memory, or operating a disk drive. They insulate programmers from the hard, complicated job of interacting with the machine at the deepest level of the naked hardware. These routines were written to expect much of the data that they process to be presented in a pre-defined arrangement, called a structure. There are more than 40 routines that use the Window structure, for instance; and they all expect that the window's width will be defined by a word (2 bytes) appearing 8 bytes from the start of the structure.

Icon .info File Structures

Below are abbreviated listings of the five structures that are used in icon .info files. I have edited this material from Rhett Anderson's and Randy Thompson's *Mapping the Amiga*, published by Compute! Books, which I find to be a compact and handy reference. The definitive source for this information is the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*, Third Edition.

First comes the name of the structure: e.g., DiskObject. Next comes the size of the structure, in bytes. The numbers in the column under 'byte' give the position, counted in bytes, of each element in the structure. The initial byte is counted as byte 0. The names of the elements are given in their Machine Language (actually, Assembly Language) versions. The ML heading stands for Machine Language, and each element is described in ML terminology. A BYTE, as everyone knows, is 8 bits. A WORD is 2 bytes. A UWORD is an unsigned word, which always signifies a positive integer, as opposed to a signed word, which can be either a positive or a negative integer. A LONG is 4 bytes. An APTR is an address pointer: a 4-byte, or 32-bit, Amiga memory address. A STRUCT signifies a complete additional structure embedded in this structure. For instance, do_Gadget is a Gadget structure embedded in the DiskObject structure. Of course, the "do" prefix stands for "DiskObject".

DiskObject
78 bytes

byte	name	ML
0	do_Magic	UWORD
2	do_Version	UWORD
4	do_Gadget	STRUCT
48	do_Type	UWORD
50	do_DefaultTool	APTR
54	do_ToolTypes	APTR
58	do_CurrentX	LONG
62	do_CurrentY	LONG
66	do_DrawerData	APTR
70	do_ToolWindow	APTR
74	do_StackSize	LONG

DrawerData
56 bytes

byte	name	ML
0	dd_NewWindow	STRUCT
48	dd_CurrentX	LONG
52	dd_CurrentY	LONG

Gadget
44 bytes

byte	name	ML
0	gg_NextGadget	APTR
4	gg_LeftEdge	WORD
6	gg_TopEdge	WORD
8	gg_Width	WORD
10	gg_Height	WORD
12	gg_Flags	WORD
14	gg_Activation	WORD
16	gg_GadgetType	WORD
18	gg_GadgetRender	APTR
22	gg_SelectRender	APTR
26	gg_GadgetText	APTR
30	gg_MutualExclude	LONG
34	gg_SpecialInfo	APTR
38	gg_GadgetID	WORD
40	gg_UserData	APTR

Image
20 bytes

byte	name	ML
0	ig_LeftEdge	WORD
2	ig_TopEdge	WORD
4	ig_Width	WORD
6	ig_Height	WORD
8	ig_Depth	WORD
10	ig_ImageData	APTR
14	ig_PlanePick	BYTE
15	ig_PlaneOnOff	BYTE
16	ig_NextImage	APTR

NewWindow
48 bytes

byte	name	ML
0	nw_LeftEdge	WORD
2	nw_TopEdge	WORD
4	nw_Width	WORD
6	nw_Height	WORD
8	nw_DetailPen	BYTE
9	nw_BlockPen	BYTE
10	nw_IDCMPFlags	LONG
14	nw_Flags	LONG

```
        18                    nw_FirstGadget
APTR
        22                    nw_CheckMark
APTR
        26                    nw_Title
APTR
        30                    nw_Screen
APTR
        34                    nw_BitMap
APTR
        38                    nw_MinWidth                  WORD
        40                    nw_MinHeight
WORD
        42                    nw_MaxWidth                  WORD
        44                    nw_MaxHeight
WORD
        46                    nw_Type
WORD
```

Icon .info File Blueprint

Every icon .info file begins with a DiskObject structure. Therefore, the first word (2 bytes) of a valid icon .info file is always do_Magic. This so-called 'magic' number is hexadecimal E310 (decimal 58,128). Unless this word is present, the system will not recognize the file as a valid icon .info file. You will occasionally come across other .info files, named simply '.info', with no file name. The script, RecolorIcons.rexx, will ignore them. They are not icon .info files, and are outside the scope of this article.

The DiskObject structure contains an embedded Gadget structure, and two bits in the gg_Flags word of this structure define what happens when the icon is selected by clicking on it with the left mouse button. The least significant, or rightmost, bit of the word is counted as bit 0. If bit 1, the GADGHIMAGE bit, is set (to 1), a completely different image will be displayed, and there will be two complete Image structures in the .info file. Otherwise, there will be only one. For example, many Drawer icons have this feature, displaying an open drawer when clicked on. If bit 1 is cleared (to 0), then only a single image is used, and the appearance of the selected icon is determined by bit 0. If bit 0 is cleared (GADGHCOMP), the image will be complemented when the icon is selected. Black and white will be reversed, and so will grey and blue. If bit 0 is set (GADGBACKFILL), the selected image will also be complemented, but any background area that has been changed from grey to blue will be flooded with grey, so that the image is not set against a blue background.

There are three types of icons which open windows when double-clicked on—Disk, Drawer, and Trashcan icons. The system requires a NewWindow structure to provide it with the data it needs to open a window. In an icon .info file, this structure comes embedded in a DrawerData structure. Whether or not the icon opens a window is specified by the first byte of the do_Type word of the DiskObject structure, immediately following the embedded Gadget structure. I don't know what the second byte of this word is used for. If the first byte has a value of 1 (WBDISK), 2 (WBDRAWER), or 5 (WBGARBAGE), the icon opens a window; otherwise it does not. Of course, the 'WB' stands for 'WorkBench'. If the icon opens a window, the .info file will contain a DrawerData structure, with an embedded NewWindow structure.

At first, I assumed that if an icon did not open a window, its .info file would never contain a DrawerData structure. After all, why would an .info file contain a structure which will never be used? Please don't expect me to answer that question. I haven't the slightest idea. All I know is that sometimes it does. I discovered this when early versions of RecolorIcons.rexx, based on my unassailable logic, sometimes failed.

The do_DrawerData address pointer of the DiskObject structure seems to be a more reliable indicator of the presence or absence of a DrawerData structure. If do_DrawerData is null (all 32-bits cleared to 0), there is no DrawerData structure. Otherwise, there is one. As nearly as I can tell, when this longword is not null, it represents the actual address of the DrawerData structure in memory, before the .info file was made and saved to disk. When a program constructs an icon, the various structures and data elements required, reside in memory in no predefined relationship. The programmer can put them almost anywhere he wants. He may group them together as a matter of convenience, but he is not obliged to. When the system assembles an icon .info file, it locates all these elements, wherever they may be, and copies them to the .info file in an unvarying, arbitrary order. The do_DrawerData pointer, which points to the DrawerData structure at that moment, if it exists, is copied to the .info file as part of the DiskObject structure. If no DrawerData structure exists, this pointer will be null. When the system reads the .info file from disk to display an icon, do_DrawerData is no longer a valid address, and the only significance of the longword is whether or not it is null.

Three other address pointers are similar. When they appear in a .info file, none of their addresses are valid. gg_GadgetRender, which pointed to the first Image structure in memory, is never null, since every icon .info file must contain at least one Image structure. In memory, gg_SelectRender pointed to a second Image structure if it existed; otherwise, it is null. In RecolorIcons.rexx, I test the GADGHIMAGE bit of the gg_Flags word to see whether the icon has one image or two, but you could test gg_SelectRender for null to get the same information. In memory, ig_ImageData pointed to the initial byte of its Image structure's bitplanes. This pointer is never null, since an Image structure without image data serves no purpose.

At last, we are ready to describe the arrangement of structures and data elements in an icon .info file. The first element is always a DiskObject structure, with its embedded Gadget structure. If a DrawerData structure (with its embedded NewWindow structure) exists, it immediately follows the DiskObject structure. The first Image structure immediately follows the DrawerData structure, if it exists. If not, it immediately follows the DiskObject structure. The image data for the first image, organized in bitplanes, immediately follows the first Image structure. If this is a dual-image icon (GADGHIMAGE), the second Image structure comes right after the last bitplane, and is immediately followed by its bitplanes of image data. Following these, there may or may not be additional data elements in an icon .info file, but none of them concerns us.

Rule Breakers

The Amiga ROM Kernel Reference Manual: Libraries, Third Edition, specifies on page 353 that the image depth (ig_Depth, the number of bitplanes) of an icon image must be two. However, some icons break this rule. There exist illegal icons with only 1-bitplane per image, or with more than two. The system will display these icons, and usually you cannot tell by looking at them that they are illegal. However, a program intended to recolor icons cannot assume that the images will always be 2-bitplanes deep, and RecolorIcons.rexx does not. On the same page, the manual also specifies that ig_PlanePick must be 3 (bits 0 & 1 both set, bits 2 through 7 cleared), signifying that only Bitplane0 and Bitplane1 are to be used. This rule is also sometimes broken.

Bitplanes

You probably know already that the earliest Amiga models up to the Amiga 3000 use 32 color registers to hold the colors that are displayed on any screen. Later models, such as the 1200 and 4000, equipped with the new custom chips and the latest operating system, have 256 color registers. Each of these 32 registers is identified by a number from 0 to 31. In binary, these color register numbers range from 00000 to 11111. Any color register can be identified by a 5-bit number. Each pixel displayed on the screen is assigned one of these numbers, and these data are organized in bitplanes. A screen that uses only two colors needs only one bitplane, Bitplane0. A 32-color screen uses five.

A two-color screen uses only color registers 0 and 1, so the color register number of any pixel can be expressed in only one binary bit. Imagine a two-color image which is 19 pixels wide by 10 lines high. This image can be represented by a bitplane in which each horizontal line of the image corresponds with 19-bits in the bitplane, giving a total of 190-bits for the entire image. However, 19-bits is an inconvenient amount of data to fetch from memory. Since the earliest Amigas fetch one word, or 16-bits, of data from memory at a time, bitplanes were designed so that each line of an image is represented by some whole number of words in the bitplane, called the word width. One word, or 16-bits, is not enough to represent 19 pixels, so each line of a 19-pixel-wide image is represented by two words, or 32-bits, in the bitplane. The unused bits in each line are ignored by the system, and are conventionally cleared to 0. Our 19-bit by 10-line two-color image is represented by a bitplane that is 20 words, or 320-bits, long.

Conventional Workbench screens use two bitplanes, and can display four colors, so all legal icons so far also have two bitplanes and four colors. The image data for one image of a legal icon consist of Bitplane0, followed by Bitplane1. The binary number identifying the color register assigned to any pixel is formed by writing the bit from Bitplane1 for that pixel, then writing the bit from the same position in Bitplane0. Below is a line of decimal digits representing the color register number of each pixel in one line of a 19-pixel-wide icon image, followed by its equivalent binary representation in Bitplane0 and Bitplane1:

```
0022200001312203330    Color Register Numbers

00000000011100011100000000000000 Bitplane0
00111000001011011100000000000000 Bitplane1
```

The hexadecimal equivalents are 0071C000 for Bitplane0, and 382DC000 for Bitplane1.

Notice what happens when you reverse the order of the bitplanes:

```
00111000001011011100000000000000 New Bitplane0
00000000011100011100000000000000 New Bitplane1

0011100002321103330     Bitplanes reversed
0022200001312203330     Original order
```

The line of decimal digits just under the reversed bitplane lines contains the new color register number for each pixel. For easy comparison, I've put the original line of color register numbers just under it. Wherever there was a 1 in the original line, there is now a 2 in the new one, and wherever there was a 2 in the original, there is a 1 in the new one. The 0s and 3s remain unchanged. Here is why it happens. The color register number for the third pixel, for instance, is expressed in binary form by writing the third bit from Bitplane1, then the third bit from Bitplane0. This gives 01, which is the binary equivalent of decimal 1. If you reverse the bitplanes, the same procedure will give binary 10, which is decimal 2. However, if both bits are either 0 or 1 to begin with, you get the binary number 00 (decimal 0), or 11 (decimal 3), whether the bitplanes have been reversed or not. The standard default colors on the new Workbench screens for color registers 0, 1, 2, and 3, are grey, black, white, and light blue. That is why IconEdit and RecolorIcons.rexx switch the black and white colors, but leave grey and blue as they were, when they recolor icons.

RecolorIcons.rexx recolors icons by reversing the order of Bitplane0 and Bitplane1 in each icon image. To do this, it must compute the length in bytes of each bitplane, which is the word width, or number of words per image line, times two bytes per word, times the height, or number of lines in the image. On page 226, the *Amiga ROM Kernel Reference Manual: Libraries*, Third Edition, gives this formula for computing the word width:

```
WordWidth = ((Width + 16) / 16)
```

I take it that the slash (/) character represents an integer division operator, since that is the only way the formula makes sense. In that case, this formula is incorrect. It produces a word width value which is one too large in those cases where the Width is an exact multiple of 16 pixels. This becomes obvious if you work through the formula with a Width value of 0.

I use this slightly amended formula:

```
WordWidth = ((Width + 15) / 16)
```

So far, it has always worked.

RecolorIcons.rexx

I won't provide a fully detailed, ARexx tutorial-style explanation of the script, since that would take up too much space. If you have at least a beginner's knowledge of ARexx, you should have no trouble following along. If you need some help getting started with the language, you might check out Merrill Callaway's excellent tutorials/articles in back issues of this magazine; his *The ARexx Cookbook*; and the Abacus book, *Using ARexx on the Amiga*, by Chris Zamara and Nick Sullivan. I have a couple of other references, but I've used these the most.

Here is how the script works. The first two code lines:

```
PARSE ARG infofile
infofile=STRIP(infofile)
```

get all the characters the user typed in after 'rx RecolorIcons' when he started the script, and remove any unwanted spaces from each end of the string. If the string has any characters left, the next 15 code lines parse the remainder into a pathname and a file name, change the script's current directory to the one specified by the pathname, if it exists, using the PRAGMA() function, and retrieve its name as a string called 'curdirpath'. Otherwise, the script just retrieves the pathname of the current directory it inherited from the Shell.

If a valid .info file name, having a .info suffix, exists, the script calls the internal function, SwapBitplanes(), which reverses the order of the first two bitplanes of one or both images, if it can. If there is no file name, the script lets the user choose whether to recolor all the icons in the current directory only, or also all icons in all directories contained in the current directory. Then it calls the procedure, Recolor().

Recolor() retrieves the pathname (curdirpath) passed to it, then once again uses the PRAGMA() function to change to the specified directory. The first time it does this, it doesn't really change anything, since the directory it changes to was the script's (not necessarily the Shell's) current directory already.

Next, it uses the SHOWDIR() function to get a list called 'files' of all the files in the current directory, only some of which will be .info files. It uses the slash (/) character to separate file names, since this reserved character is never used within a file name. Some other characters, such as the colon (:), would also work.

Unlike some C directory commands, the SHOWDIR() function does not allow the use of 'wild cards', (e.g., 'List #? info'), so you can't use it to retrieve a list of only .info files. Instead, .info file names must be located within and copied from the complete list of all the file names in the target directory that the function produces.

I wrote the program so that I could recolor many icons in one batch, with a minimum of effort. I've used it to recolor over 2000 icons on dozens of floppies, and so far the script has digested every icon I've fed it, including some that break Commodore's programming rules.

I wanted to retrieve and print the .info filenames in their original mixed case form. The system creates .info file names using the lower case form of the suffix, but users sometimes edit them. Since I don't know a way to make the PARSE instruction case-insensitive, the script emulates this capability using the INDEX() and LASTPOS() functions. It makes an UPPER CASE copy called 'ufiles' of the string 'files'. Starting from the beginning, the script searches ufiles for the substring, '.INFO/'. Then it searches backwards for the first slash character. This gives the position of the .info file name within the string 'ufiles', which is the same position that the file name has in 'files', from which the original mixed case file name is extracted. Including the period and the trailing slash in the parsing substring '.INFO/', prevents errors when odd file names such as 'Information.informers.info' appear in the file list, since slash characters occur only at the beginning and end of each file name. As each .info file name is retrieved, the script calls the internal function, SwapBitplanes(), which recolors the icon by reversing the order of the first two bitplanes in each image, until all the .info files in the current directory have been processed.

If the user has chosen to also recolor the icons in all child directories, the script uses the SHOWDIR() function to make a list of all the directories in the current directory, once again using the slash character to separate the names. This time, since all of the names in the list produced by SHOWDIR() are needed, so that the slash character can be used by itself as a parsing marker, the PARSE instruction is able to extract mixed case directory names. As each directory name is retrieved, the full pathname (nextdirpath) is made by adding the new directory name to the current pathname. Recolor() then calls itself, passing the new directory pathname to the new invocation of the Recolor() procedure. The new invocation of Recolor() recolors all of the icons in its new current directory, then makes a list of all the directories it finds there, and calls Recolor() again for each of them.

This is the most common form of recursion. Each invocation of an ARexx procedure maintains its own separate table of internal variables, which are not accessible to other parts of the script, not even to other invocations of the same procedure, unless they are explicitly EXPOSEd. However, these internal variables are made available to other internal functions called by the original procedure. The Recolor() procedure calls itself over and over, until it has reached every directory contained in the original, and though each invocation of the procedure uses several variables such as 'curdirpath', 'files', and 'newdir', which have the same names, their contents are unique to each invocation of Recolor(). The only exceptions are the EXPOSEd variables, 'choice', and 'onlyicon'.

Recursion is subtle and a bit tricky to handle, but it is very powerful. The DO loop that retrieves each new directory name, makes the new directory pathname, and performs the recursion, consists of only five lines. These few lines contain all of the logic by which the script snoops into every nook and cranny of the directory tree contained in the original directory, no matter how many branches it has, and tracks down every last .info file. If you are interested in recursion, Merrill Callaway described another type in his article, Recursive Function Calls in ARexx, in the V.7.3 issue of *Amazing Computing*, in which he presented a program that uses recursion to solve the so-called "Coconut Problem."

The internal function, SwapBitplanes(), and three subsidiary functions, do the actual work of recoloring the icons. SwapBitplanes() first opens the specified .info file if it can, then reads the first word to see if it is the 'magic' number, hex E310, which identifies the file as a true icon .info file. Then, it reads the gg_Flags word and tests the GADGHIMAGE bit (bit 1), to see if this is a dual-image icon. Next, it checks the first byte of the do_Type word, to see if this icon opens a window when double-clicked on. Originally, the idea was to see whether or not the .info file contains a DrawerData structure, so that the script will be able to find the beginning of the first image structure. Since it is possible for an icon .info file which does not open a window to contain a DrawerData structure, this test is unreliable and unnecessary; but I decided not to remove it, in case I ever want to use the information it provides (the icon type) for another purpose. Next, SwapBitplanes() finds out for sure whether or not a DrawerData structure exists by testing the do_DrawerData address pointer for null.

Since SwapBitplanes() now knows whether or not the .info file contains a DrawerData structure, it can figure out how many bytes to SEEK forward to arrive at the third word of the first Image structure. It calls ReadImageStructure(), which reads the ig_Width, ig_Height, and ig_Depth words, and uses the first two to compute the word width, then the bitplane length.

If the image depth (the number of bitplanes) is two or more, SwapBitplanes() now calls Swap2(). Swap2() SEEKs forward to the start of the image data, reads the first two bitplanes, and writes them back in reverse order. This switches the colors black and white, but not grey and blue, for reasons I explained under Bitplanes.

If the image depth is only 1, the icon image consists of only the first two Workbench colors, grey and black. SwapBitplanes() calls Invert1(), which swaps these two colors by switching all the 0 bits of the single bitplane to 1, and all the 1 bits to 0. The microprocessor is provided with a NOT instruction, which does just that, but ARexx has no equivalent function. However, ARexx's BITXOR() function can be made to give exactly the same result. This function examines two strings and compares them bit by bit. If both bits in the same position are 0s, or both are 1s, the function puts a 0 in the equivalent position of the output string. If either bit is 0, and the other is 1, a 1 is placed in the output string. If you use BITXOR() to compare any string with a string of the same length in which all bits are set, the output string will have a 0 in each position where the original string has a 1, and a 1 in each position where it has a 0. If you provide an empty string ('') as the second string, and also provide a pad byte in which all bits are set (hex FF), BITXOR() will automatically generate a comparison string of the correct length in which all bits are set. This gives exactly the same result that the microprocessor's NOT instruction would produce.

Invert1() swaps the two colors (grey and black) of illegal icons with single-bitplane images. This certainly changes the appearance of the icon, but it is not the same as swapping a legal icon's black and white colors. Chances are you won't come across any single-bitplane icons, but if you don't want the script to do this, you can switch off Invert1() by changing the value of Invert to 0 at the start of SwapBitplanes().

If the icon has a second image, SwapBitplanes() SEEKs past any unused bitplanes from the first image to the third word of the second image structure, and processes the second image as already described. Finally, SwapBitplanes() closes the .info file and RETURNs. When the last icon has been recolored, the script EXITs.

ARexx Procedures

I just want to point out a subtlety in the way EXPOSEd variables work in ARexx procedures. Understanding it can save you some pain when you develop your own ARexx scripts. The Recolor() procedure uses two EXPOSEd variables, 'choice', and 'only1icon'. 'choice' is used in Recolor(), but 'only1icon' is not. So why does 'only1icon' need to be declared as an EXPOSEd variable?

The answer is that, if it is not, the internal function, SwapBitplanes(), which uses 'only1icon', won't work properly when it is called by Recolor(). 'only1icon' is assigned a value near the start of the script. When SwapBitplanes() is called from the early part of the script, not from Recolor(), 'only1icon' is available, and SwapBitplanes() works fine. But when it is called from Recolor(), SwapBitplanes() behaves as though it is part of the Recolor() procedure, so that it is protected from variables which are not available to Recolor(). Since 'only1icon' is assigned a value outside of both Recolor() and SwapBitplanes(), it must be EXPOSEd by Recolor() to make it available to SwapBitplanes(). This nuance tripped me up as I was developing RecolorIcons.rexx.

If you want to check this out for yourself, here's how. Edit RecolorIcons.rexx, removing 'only1icon' from the list of EXPOSEd variables in Recolor(). Now, run RecolorIcons.rexx on a directory containing an .info file named simply '.info', with no preceding file name. When the script offers you a choice, enter either 1 or 2. You will find such files in the root directories of many disks, such as the Fred Fish series. These files are not true icon .info files, so they cannot be recolored. Instead of reporting this information and continuing, ARexx will generate an error, complaining that 'only1icon' has not been assigned a Boolean value, and halt the script. If you try the same thing with an unedited version of RecolorIcons.rexx, it will work fine.

Solving the Next Half

Well, that takes care of all the old icons. Once you have finished using the script, your Workbench should be looking better. But at this point, we have solved only the first half of the problem. As long as you continue to use old programs which generate icons, such as WordPerfect, you will keep on creating icons that need to be recolored. Even if you can do the job quickly, why should you have to bother? Of course, one solution would be to replace all your old software, but that would be a lot of trouble and expense.

Fortunately, there is a better way. It is fairly easy to customize programs so that, within limits, they will generate almost any icons you like. In my next article, *Re Color Revisited*, I'll provide several ARexx scripts that you can use to solve this problem permanently for WordPerfect and some other programs, and I'll describe a method which you will be able to use to customize most of your other old software. If the idea appeals to you, don't throw away this magazine. You will need some of the information in this article to understand the next one. In the meantime, I hope you enjoy using RecolorIcons.rexx.

✓

Listing

```
/****************** RecolorIcons.rexx
******************/
/*
*/
/*              Copyright 1993 by Dave Senser
*/
/*
*/
/*                        May 1993
*/
/*
*/
/* Please keep my name, this notice, and all comments
*/
/* intact in any distribution of this program.
*/
/*
*/
/* This AREXX script recolors old icons made using
*/
/* programs written for pre-O.S. 2 versions of the
*/
/* operating system, reversing their black and white
*/
/* colors, so that they will display correctly on newer
*/
/* versions of the O.S.
*/
/*
*/
/
*****************************************************************/


/********************** U s a g e
*********************/
/*
*/
/*   Assuming that ARexx is properly set up on your
*/
/* system, and that this script is in your REXX:
*/
/* directory, CD a Shell to a directory containing
*
/* icon(s) you want to recolor. To recolor a single
*/
/* icon, enter, for instance:
*/
/*
*/
/*         rx RecolorIcons MyFile.info
*/
/*
*/
/*   To recolor several icons, omit the file name, and
*/
/* follow the prompts.
*/
/*
*/
```

```
/
*****************************************************************/

/*  Recolors .info files by swapping two bitplanes of */
/* image data, reversing black & white colors. If .info
*/
/* file is for a dual-image type of icon, swaps second */
/* pair of bitplanes, as well. */

/*  This version fixes problems with some icons which */
/* break the rules, such as the one that looks like */
/* a book with the title 'Doc File' on the cover, */
/* that appears on many Fred Fish disks. This icon */
/* uses 3 bitplanes instead of two, so if the first image
*/
/* structure specifies a depth of 3 or more, the script */
/* swaps only the first two bitplanes, and SEEKs past */
/* the additional bitplane(s) before reading the second */
/* image structure (if there is one), then swapping two */
/* more bitplanes. */

/*  The Third Edition of the ROM Kernel Manual, LIBRARIES
*/
/* volume, specifies on page 353 that the image depth MUST
*/
/* be 2, and PlanePick MUST be 3. This icon uses an image
*/
/* depth of 3, and sets PlanePick to 7 (all 3 least sig-
*/
/* nificant bits set, signifying that all 3 planes are to
be */
/* used), breaking two of the rules for icons designed for
*/
/* pre Version 2 OS's. Oddly enough, only the first two
*/
/* bitplanes for each image contain image data. Each
third */
/* plane is null. */

/*  This version also works with icons which use */
/* single-bitplane images, which are also illegal. */
/* Such images don't have two bitplanes to swap, so */
/* script inverts the single bitplane, flipping ones */
/* to zeros, and zeros to ones, which swaps the two */
/* colors. */

/*  This version also works with icons, other than */
/* WBDISK, WBDRAWER, or WBGARBAGE, which have a */
/* DrawerData structure. Script checks do_DrawerData */
/* longword for null, and, if not null, SEEKs past */
/* DrawerData structure to get into first image structure.
*/


PARSE ARG infofile            /* Get file name, if any */
infofile=STRIP(infofile)      /* Strip spaces from each
end */

/* Check to see if user has specified a different */
/* directory. If so, obtain pathname and file name, */
/* and change to new directory, if possible. */

slash=LASTPOS('/',infofile)
colon=LASTPOS(':',infofile)
IF slash>0 THEN DO
  newdir=LEFT(infofile,slash)
  infofile=RIGHT(infofile,LENGTH(infofile)-slash)
  IF PRAGMA('d',curdirpath)='' THEN SIGNAL BadPath
   END
ELSE IF colon>0 THEN DO
  curdirpath=LEFT(infofile,colon)
  infofile=RIGHT(infofile,LENGTH(infofile)-colon)
  IF PRAGMA('d',curdirpath)='' THEN SIGNAL BadPath
   END

curdirpath=PRAGMA('d')  /* Get full pathname of current
directory */
IF RIGHT(curdirpath,1)=':' THEN separator=''
ELSE separator='/'

IF infofile='' THEN onlyicon=0
ELSE onlyicon=1               /* If file name, set flag
*/

IF onlyicon THEN DO
  IF UPPER(RIGHT(infofile,5))~='.INFO' THEN DO
      SAY
     SAY "File name must have a '.info' suffix. "||,
         "Sorry, no can do!"
      EXIT 20                      /* Quit */
      END
   SAY
  CALL Swapbitplanes()
   SAY
  SAY "That gets it.  So long!"
   EXIT 0                         /* Quit */
   END
ELSE DO UNTIL choice=1 | choice=2
   SAY
  SAY "Your current directory is:"
   SAY
  SAY curdirpath
   SAY
  SAY "  You can recolor ALL ICONS IN THIS DIRECTORY
ONLY (1),"
  SAY "or also ALL ICONS IN ALL CHILD DIRECTORIES (2)."
   SAY
  OPTIONS PROMPT "Enter 1, 2, or Q to quit:  ==> "
   PULL choice
  choice=LEFT(COMPRESS(choice),1)
  IF choice='' |choice='Q' |choice=XXC(1B) THEN EXIT 0
   END                       /* Escape */  /* Quit */

IF choice=2 THEN DO UNTIL YesNo='Y' | YesNo='N'
   SAY
  SAY "  Script will recolor ALL ICONS IN THIS DIREC-
TORY,"
  SAY "AND ALSO IN ALL CHILD DIRECTORIES."
   SAY
  OPTIONS PROMPT "  Is this what you want?  (Y/N) ==> "
   PULL YesNo
  YesNo=LEFT(COMPRESS(YesNo),1)
   END

IF YesNo='N' THEN DO
   SAY
  SAY "No icons recolored."
   EXIT 0                          /* Quit */
   END

SAY
CALL Recolor(curdirpath)

SAY
SAY "That does it.  So long!"
EXIT 0                          /* Quit */

/***** Internal functions follow *****/

Recolor: PROCEDURE EXPOSE choice onlyicon

  PARSE ARG curdirpath          /* Get new directory
pathname */
  IF RIGHT(curdirpath,1)=':' THEN separator=''
  ELSE separator='/'

  x=PRAGMA('D',curdirpath)     /* Change to new direc-
tory */

  files=' '||(SHOWDIR('','F',' /'))||' '  /* Get file list
*/
  ufiles=UPPER(files)           /* Make UPPER CASE copy
*/

   info=1
   DO WHILE info>0              /* Extract .info file
names */
    info=INDEX(ufiles,'.INFO ',info)
      IF info>0 THEN DO
       slash=LASTPOS('/',ufiles,info)
         info=info+5
      infofile=SUBSTR(files,slash+1,info-slash-1)
        CALL SwapBitplanes()     /* Swap 2nd & 3rd colors
or swap */
         END                     /* -- 2 colors of only
bitplane */
      END

  DROP files
  DROP ufiles

  IF choice=2 THEN DO
    dirs=SHOWDIR('','d','/')||'/'     /* Get directory
list */

     DO WHILE dirs~='/'
        PARSE VAR dirs newdir '/' dirs  /* Build new
pathname & -- */
     nextdirpath=curdirpath||separator||newdir
        CALL Recolor(nextdirpath)       /* -- Recolor new
directory */
         END

      END
  RETURN

SwapBitplanes:

  Invert=1  /* 1 inverts single-bitplane images, 0
switches off */

  IF ~OPEN('patchfile',infofile) THEN DO /* Try to open
specified file */
    SAY "Can't find '"curdirpath||separator||infofile||,
         "'.  Sorry, no can do!"
      IF onlyicon THEN EXIT 20     /* Quit */
      RETURN
      END

  magic=READCH('patchfile',2)    /* Start of DiskObject
structure */
```

```
  IF magic~=X2C(E310) THEN DO     /* If not icon .info
file, quit */
   SAY '"'curdirpath||separator||infofile||,
       '" is not a true icon .info file.'
   x=CLOSE('patchfile')
     IF onlyIcon THEN EXIT 0     /* Quit */
     RETURN
     END

  /* If you don't want .info files listed as they are
processed, */
  /* disable next line. */

 SAY 'Recoloring 'curdirpath||separator||infofile

  x=SEEK('patchfile',14)          /* Gadget structure
embedded - */
  flags=READCH('patchfile',2)     /* - in DiskObject
structure */
  GADGHIMAGE=BITTST(flags,1)      /* Dual-image icon?
*/

  WBDISK=1
  WBDRAWER=2
  WBGARBAGE=5
  x=SEEK('patchfile',10)          /* DiskObject struc
ture */
  type=C2D(READCH('patchfile',1)) /* Does icon open
window?? */
  IF type=WBDISK | type=WBDRAWER | type=WBGARBAGE THEN
window=1
  ELSE window=0
  x=SEEK('patchfile',17)          /* DrawerData struc-
ture exist?? */
  do_DrawerData=C2D(READCH('patchfile',4))

  /* If icon opens a window when double-clicked on, */
  /* or even if it doesn't open a window, but */
  /* contains a DrawerData structure, then SEEK */
  /* past DrawerData structure and into Image structure.
*/
  /* Else, just SEEK into Image structure. */

  IF window | do_DrawerData~=0 THEN
x=SEEK('patchfile',68)
  ELSE x=SEEK('patchfile',12)     /* SEEK to 3rd word of
1st - */
  CALL ReadImageStructure()       /* - Image structure,
then read */
  IF depth>1 THEN CALL Swap2()
  ELSE IF Invert THEN CALL Invert1()
   ELSE DO
   x=SEEK('patchfile',bpLength+10)
   SAY curdirpath||separator||infofile,
       "first single bitplane not inverted."
     END

  /* If icon uses 2 images, then swap second pair of */
  /* bitplanes (or invert second single bitplane), also.
*/

  IF GADGHIMAGE THEN DO           /* Do, if second image
exists */
   IF depth>1 THEN x=SEEK('patchfile',(depth-
2)*bpLength+4)
    ELSE x=SEEK('patchfile',4)    /* SEEK to 3rd word of
2nd - */
    CALL ReadImageStructure()     /* - Image structure,
then read */
    IF depth>1 THEN CALL Swap2()
    ELSE IF Invert THEN CALL Invert1()
   ELSE SAY curdirpath||separator||infofile,
          "second single bitplane not inverted."
    END

  x=CLOSE('patchfile')            /* Close patched file
*/
  RETURN

ReadImageStructure:

 width=C2D(READCH('patchfile',2)) /* 3rd word of Image
structure */
 height=C2D(READCH('patchfile',2))
 depth=C2D(READCH('patchfile',2))
 wordWidth=(width+15)%16
 bpLength=wordWidth*height*2
  RETURN

Swap2:
                                  /* SEEK past end of Image
structure */
 startBitplane0=SEEK('patchfile',10)
 a=READCH('patchfile',bpLength)   /* Read 2 bitplanes */
 b=READCH('patchfile',bpLength)
 x=SEEK('patchfile',startBitplane0,'begin')
  x=WRITECH('patchfile',b)        /* Write back 2
bitplanes - */
  x=WRITECH('patchfile',a)        /* - in reverse order
*/
  RETURN

Invert1:
                                  /* SEEK past end of Image
structure */
 startBitplane0=SEEK('patchfile',10)
 a=READCH('patchfile',bpLength)   /* Read only bitplane
*/
  a=BITXOR(a,,'FF'x)              /* Invert */
 x=SEEK('patchfile',startBitplane0,'begin')
  x=WRITECH('patchfile',a)        /* Write back */
  RETURN

BadPath:

  SAY
  SAY "Bad pathname. Try again."
  EXIT 20                         /* Quit */
```

Complete source code can be found on the AC's TECH disk.

Please write to:
Dave Senger
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722

SUBSCRIBE!

YES! *The "Amazing" AC publications give me* ***3 GREAT*** *reasons to save!*
Please begin the subscription(s) indicated below immediately!

Name ______________________________

Address ______________________________

City ______________________ State ______ ZIP ________

Discover Visa MC # ______________________________

Expiration Date ________ Signature ______________________

Please circle to indicate this is a **New Subscription** or a **Renewal**

MasterCard VISA DISCOVER

Call now and use your Visa, Master Card, or Discover or fill out and send in this order form!

1 year of **AC**	12 big issues of Amazing Computing! Save over 43% off the cover price!	US $27.00 ☐ Canada/Mexico $34.00 ☐ Foreign Surface $44.00 ☐
1-year SuperSub	AC + AC's GUIDE—14 issues total! Save more than 45% off the cover prices!	US $37.00 ☐ Canada/Mexico $54.00 ☐ Foreign Surface $64.00 ☐
1 year of AC's TECH	4 big issues of the FIRST Amiga technical magazine with Disk!	US $43.95 ☐ Canada/Mexico $47.95 ☐ Foreign Surface $51.95 ☐

Please call for all other Canada/Mexico/foreign surface & Air Mail rates.
Check or money order payments must be in US funds drawn on a US bank, subject to applicable sales tax.

ORDER!

NAME ______________________________

ADDRESS ______________________________

CITY ______________________ STATE ______ ZIP ________

__DISCOVER __VISA __M/C # ______________________________

EXPIRATION DATE ____________ SIGNATURE ______________________

Amazing Computing Back Issues: $5.00 each US, $6.00 each Canada and Mexico.
$7.00 each Foreign Surface. Please list issue(s) ____________________

Amazing Computing Back Issue Volumes:
Volume 1-$19.95* Volume 2, 3, 4, 5, 6, or 7—$29.95* each
*All volume orders must include postage and handling charges. $4.00 each set US, $7.50 each set Canada and Mexico, and $10.00 each set for foreign surface orders. Air mail rates available.

MasterCard VISA DISCOVER

AC's TECH AMIGA — Single issues just $14.95! Pick any 4 issues for only $45.00!
V1.1 (Premier), V1.2, V1.3, V1.4, V2.1, V2.2, V2.3, V2.4, V3.1, V3.2

Freely Distributable Software – Subscriber Special (yes, even the new ones!)

1 to 9 disks	$6.00 each
10 to 49 disks	$5.00 each
50 to 99 disks	$4.00 each
100 or more disks	$3.00 each

$7.00 each for non subscribers **(three disk minimum on all foreign orders)**

Amazing on Disk:

AC#1	Source & Listings V3.8 & V3.9	AC#2	Source & Listings V4.3 & V4.4
AC#3	Source & Listings V4.5 & V4.6	AC#4	Source & Listings V4.7 & V4.8
AC#5	Source & Listings V4.9	AC#6	Source & Listings V4.10 & V4.11
AC#7	Source & Listings V4.12 & V5.1	AC#8	Source & Listings V5.2 & 5.3
AC#9	Source & Listings V5.4 & V5.5	AC#10	Source & Listings V5.6 & 5.7
AC#11	Source & Listings V5.8, 5.9 & 5.10	AC#12	Source & Listings V5.11, 5.12 & 6.1
AC#13	Source & Listings V6.2 & 6.3	AC#14	Source & Listings V6.4 & 6.5
AC#15	Source & Listings V6.6, 6.7, 6.8, & 6.9		

Please list your Freely Redistributable Software selections below:

AC Disks ______________________
(numbers 1 through 15)

AMICUS ______________________
(numbers 1 through 26)

Fred Fish Disks ______________________
(numbers 1 through 860)

Complete Today, or telephone 1-800-345-3360 now!

You may FAX your order to 1-508-675-6002

Please allow 4 to 6 weeks for delivery of subscriptions in US.

(Domestic and Foreign air mail rates available on request)

Check or money order payments must be in US funds drawn on a US bank, subject to applicable sales tax.

Have Your Own Custom 3-D Graphics Package

Part I of this article gave the derivation of a z-buffer algorithm for rendering objects defined by points. It gave programs for generating, placing and rendering floors, shadows, and boxes defined by points. Part II gives the derivation of a lighting algorithm for objects defined by points, and lists programs for lighting, testing the effect of a light on an object, setting an object's color, extracting a palette from an image, and generating objects of revolution. Part II includes lighting code for the floor and the cube programs listed in Part I. Part II also describes a Fine Arts coloring theory that simplifies 3-D graphics coloring.

World Space: Lighting and Color

The world space of these programs was described in Part I as real-numbered left-handed coordinate space. The world space is assumed lit by ambient light which may be chosen as low or a bright as you please. All the objects have their shadow color to start, becoming lighter when lit.

The Angle of Incidence

When light strikes an object the amount of light the object receives depends on the angle of the striking light ray. The more head on, or perpendicular, the strike, the more light the object gets. The more glancing the strike, the less light the object gets. This striking angle is called the 'angle of incidence.' It can be measured by comparing the angle the light ray makes with the 'normal line', or perpendicular, to the object's surface. See Illustration II.1 (a)

Since the object has no surface—it is all points—using the 'normal' line stretches the mathematical meaning of "the normal to a surface at a point on the surface" but it actually works quite well. For a sphere the line through the object's center and a point on the surface is perpendicular to the surface at that point. As an object's form deviates from spherical this will be less and less accurate. We can, nevertheless, within the limitations of these two considerations, base a lighting algorithm on the angle between the light ray and the 'normal' line.

Illustration II.1 (a) shows a ray from the light, PL, to the object, PO. Further down in the object is its center, PC. The normal line to the object, at the point PO, is the line from the center, PC, through PO.

The equation for the angle between two directed lines will give the 'angle of incidence.' The directed lines to use in this equation are Line1, from PO to PL, and Line2, from PO to PC. Note that these directed lines are not the same as the light ray and the normal line. See Illustration II.1 (a).

The equation is:

```
     cos(theta) = cos(alpha1)*cos(alpha2) +
cos(beta1)*cos(beta2) + cos(gamma1)*cos(gamma2)
```

Where, for Line1:

```
            cos(alpha1) = (xl-xo)/d1
            cos(beta1) = (yl-yo)/d1
            cos(gamma1) = (zl-zo)/d1
      d1 = SQRT( SQR(xl-xo) + SQR(yl-yo) + SQR(zl-zo)).
```

For Line2:

```
            cos(alpha2) = (xc-xo)/d2
```

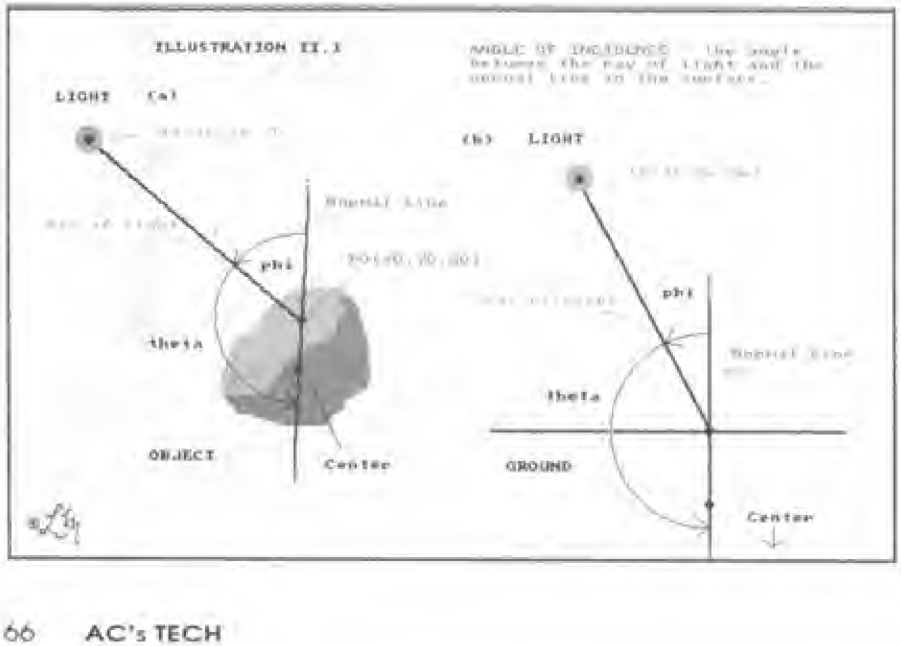
... and so on.

Detailed derivations can be found in Trigonometry and Analytic Geometry texts.

The equation yields cos(theta). Then:

```
            theta = arccos(theta).
```

The 'angle of incidence,' however, is phi not theta. Phi equals 180 degrees minus theta. The amount of light falling on an object at a point is proportional to phi. The smaller phi the more directly the light hits the object. The larger phi the



Part II—Adding the finishing touches to your personal 3-D software

by Laura M. Morrison

more glancing the ray of light hitting the object.

Calculating the amount of light falling on a point by 'angle of incidence' constructed using the 'center' point becomes less meaningful as the shape of the object deviates from the shape of a sphere. The artist, however, can compensate for this by selecting an appropriate 'center' points. The center of an object can be calculated exactly, or selected to an end of getting a particular artistic effect. An object can also be divided, several 'centers' selected, and each subset of the object's points processed separately. For example, to enhance their form, the blossoms in Illustration II.4 were processed separately, and lighted each with respect to its own center. The 'centers' for the boxes and the floors in Illustrations I.4 and II.4 were taken always on lines perpendicular to the side of the box or perpendicular to the floor plane. Listing 12A shows how to get the 'normal' for floors, Listing 12B shows how to get the 'normal' for the sides of box.

An algorithm for the reflection of one object onto another could be derived using the same algorithm by substituting the influencing object for the light. Radiosity effects can be similarly calculated.

The 'angle of incidence' provides a basis for decisions about the color of a lighted point. It does not, however, determine the point's color. You, the artist, must decide how to use the information and your decisions will personalize your images.

The decisions implemented in the 'lighter' program are not the only possible. They divide the range of phi into sections, coloring the light, middle, and extreme ranges with palette colors, and intermediary ranges with checkered mixtures of the extremes. The information can be used differently for lighting different objects. Walls and grounds need gradual lighting but blocky forms, like the blossoms in Illustration II.4, are enhanced by blocky lighting.

The lighter code included with the 'lighter' program, Listing 10, has only seven divisions of phi. It was used to light the blossoms. The code for lighting floors and cubes, Listing 12-C, has 57 divisions. It was used to light the ground and wall in Illustration II.4. The lighter code for testlighter has 60 divisions. It was used to mark the divisions of phi on Sphere (e) in Illustration II.2.

The division of phi's range can be used to set the mood of an image. For example, relatively large light and dark sub-ranges give a dramatic, chiaroscuro effect (Rembrandt). Relatively larger middle-tone ranges give a flat, decorative effect (Matisse). Illustration II.2 (a)-(d) shows spheres lighted with differently grouped divisions.

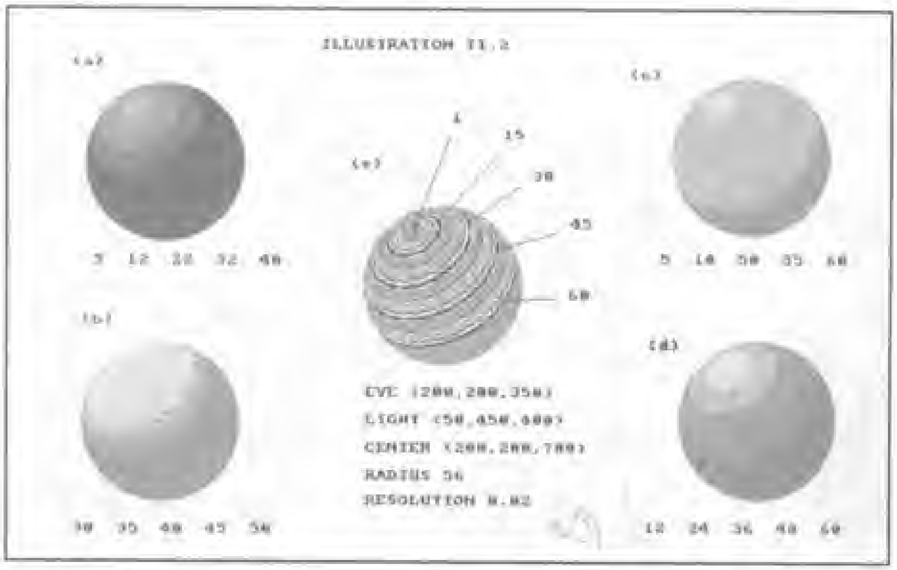
To make it easy to select and change groupings an array of 60 divisions is included in the 'lighter' code, Listing 10 (starting at line 7). Zone[] is an array of 60 equal radian divisions of phi. (Actually, 61 elements with the unused *zero* element.) Illustration II.2 (e) represents a 'lighted' sphere with each of 60 ranges of phi colored differently (modula 15). Illustration II.5(b) shows a post similarly marked.

The numbers below the spheres (a)-(d) in Illustration II.2 give the zone[] numbers used to get the distributions of light shown. For example, Illustration II.2 (a) shows a sphere lighted with phi's range divided into six parts. The highlight is the first range:

```
0 <= phi < 0.1309 radians = zone[5]
```

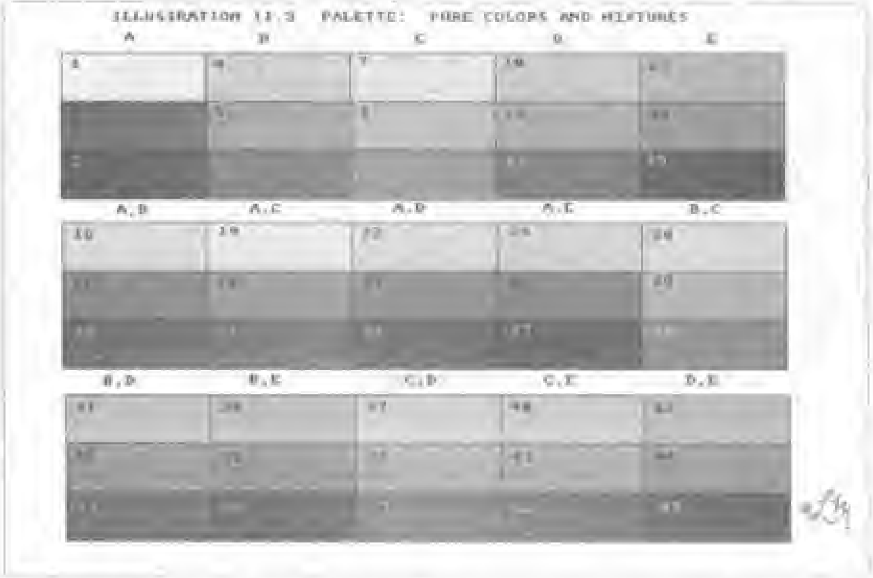
The next range where the color is a little darker is:

```
0.1309 <= phi < 0.20944 = zone[12]
```



Right: Illustration 2.2.

Opposite Left: Illustration 2.1.



The program takes care of translating to radians.

The testlighter program, Listing 11, will mark an object according to how the light strikes it. By test lighting an object to get its multiple colored representation (as sphere (e)) you can select distributions of phi that will best enhance the form of the object or the mood of the picture. The numbered lines pointing to colored rings on Sphere (e) show how, with a little practice, you will be able to "read off" division numbers on your test lighted object.

Fine Arts Coloring Theory

The Old Masters taught their students to use a limited palette of eight to twelve colors, to get new colors by mixing no more than two of these, about half and half, and, by adding to the pure colors, or the mixtures, traces of another color, to "break" the colors. They also taught their students to avoid using pure white and pure black. Some of their advise had to do with avoiding muddy colors, which characterizes paint mixtures of more than two colored paints, and does not apply to computer colors, which are pure light. Much of their theory, however, contributed to the beauty and coherence of their paintings.

The Old Masters taught that the palette's colors were best divided into tones. There might be as few as three tones, or as many as seven. Three was considered to work well. I used three for Illustration II.4. The palette colors are arranged in sequences of three tones: light, medium, and dark. In paint the tones might have been one color with white added and black added. They could be three different but related colors, or even three separate colors. If colors are changed but keep within the same classifications of tone and color temperature the result usually remains meaningful. Evidently the eye will accept a wide range of colors as belonging if only the color is consistent in tone and temperature.

Illustration II.3. gives a palette template with the colors selected for the image of the potted flowers, Illustration II.4. Three tones each of pink-red, orange-red, powder-blue, and neutral-purples, make up the 15 registers. Although black is available in color register zero it is not used in Illustration II.4. A checkered pattern makes the half-and-half mixtures. Since the blossom objects are composed of disassociated points it is difficult to color a blossom checkered. Using the sum of x and y to determine odd or even, works fairly well, however.

With only 16 color registers available mixtures are expedient, but more importantly, they are a way of insuring color coherence. Even if more registers were available, as in the new AGA chip Amigas, mixtures might still be desirable. One great advantage of the checkered pattern mixtures is that they all adjust automatically when you change a principal color.

A generally useful palette might include two sets of warm colors (yellows and reds), two sets of cool colors (greens and blues), and a set of neutrals (greys or purples). You can use Illustration II.3 as a palette template when selecting colors. When you set colors for color registers one through fifteen the mixtures that result are show underneath.

This information on Fine Arts theories and techniques can be found in *The Materials and Techniques of Painting* by Jonathan Stephens, published 1989 by Watson-Guptill Publications, New York. You are not limited to this theory. You can make up your own or use none at all. The computer, however, is ideal for applying theories, and rules applied consistently are, after all, the basis of that much valued and elusive factor of art, unity.

The Code

Programs 1 through 8 were listed in Part I. The general aspects of the code, such as using scripts, numbering files, recording program activity were described there. Following is discussion of programs 9 through 15, listed here. Additional comments are in the listings.

Top: Illustration 2.3.

Left: Illustration 2.4.

Opposite Page: Illustration 2.5.

Colorer, Listing 9, changes the register numbers of points of an object according to your input color id. It ignores the color value already stored with the points. Coloring a bunch of disassociated points 'checkered' is done by assigning one color to points with odd sums x + y and the other color to even sums x + y.

Map1[] and map2[] give the color register numbers for 45 color ids. The 45 colors are the 16 solids and their mixtures shown in Illustration II.3. Map1[id] gives the register numbers for points having even xobj + yobj. Map2[id] gives them for odd. Note that the first 15, being solids, have both numbers the same.

Colorer reads an object point's record, revises its color id, and writes it to a new file. Colorer requires a parameter file, 'coparams.' See Table 2.1. The parameter file should contain the color 'id,' not the register number of a color.

To run colorer use a script. (Given in Table2.2) At the CLI prompt, type

```
Execute color-script <object name>
```

Lighter, Listing 10, reads a point and calculates a new color for it based on the amount of light falling on it. Lighter requires a parameter file 'liparams' which contains the light position and the center point of the object. For example 'liparams' see Table2.1

The zones used for Illustration II.4 blossoms were:

```
        0 <= phi < zone[20]
zone[20] <= phi < zone[30]
zone[30] <= phi < zone[40]
zone[40] <= phi < zone[50]
zone[50] <= phi < zone[60]
zone[60] <= phi
```

Lighter program calculations are all relative to the darkest, or ambient, color of an object. That way the lighter program can be used on many different colored objects provided only that their input color are the darkest color of a range of three. This is only one possible coloring algorithm. See Listing 12A and 12B for algorithms with more divisions and a different treatment per divisions. The proportions of the mixtures were based on the remainder modula 27, which was determined experimentally.

Besides different numbers of divisions, and different sizes of divisions, you can vary the treatment within the divisions. One might even, for example, intersperse a percentage of tinting colors, add, say, 10 percent yellow to the lightest, 10 percent red to the middle, 10 percent blue to the darker tones—or any other imaginable treatment consistent with "cyberspace" optics.

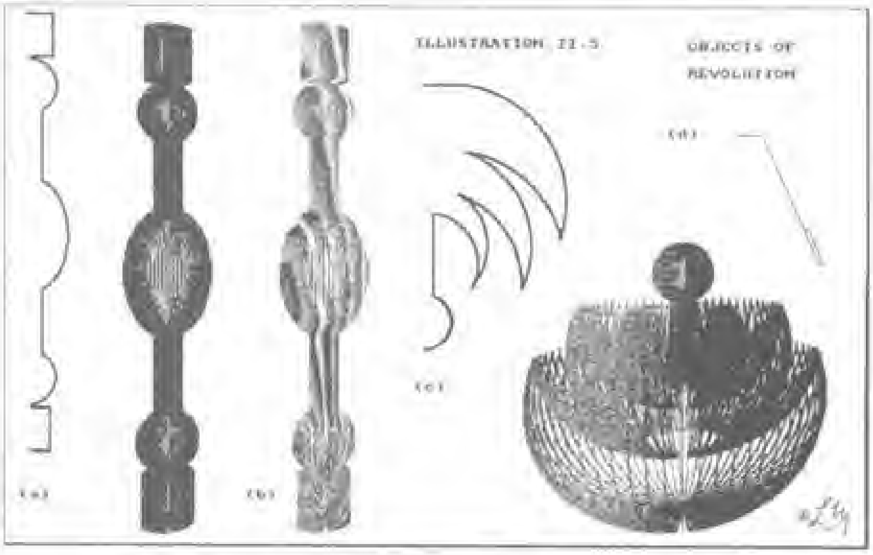
Lighter does not need the entire object to determine lighting at a point. Lighting depends only upon the position of the light, the center point of the object and the point being lit. To enhance the form of its component chunks you can break up an object into chunks, each with its own center point, and light them separately. When lighting boxes (the program for generating box points, Listing 5, was given in the Part I) using the true box center will result in a box that appears somewhat like a sphere. See box lighting code fragment, Listing 12 B. To get the correct effect with sharp edges where the planes of the box turn, you need to take a special 'center' point for each side. For example, if the box side is parallel with the x-y plane then for the 'center' use the true z center point and the current x, y coordinates you are processing so that the line from the 'center' through the point is always perpendicular to the side. If the box side is parallel with the x-z plane then use the true y center point and the current x, z coordinates so the line through the 'center' and the point on the object is always perpendicular to the side.

Similarly, the center for the floor has to be taken as some negative y value beneath the floor and the current x and z values to insure a perpendicular normal. See Illustration II.1 (b). The center is straight down, below the ground, so the equations will give a normal line perpendicular to the ground.

Fine Arts theory favored two lights only in a picture: a main light to one side and high up, and a weak fill light from the other side and lower down. The programs do not restrict the number of lights or their colors. You must, however, be able to tell the computer how to deal with each mixture of lights situation it will encounter.

To run lighter on an object of many files use a script. Enter CLI and at the prompt type:

```
Execute lighter-script <objectname>
```



Testlighter, Listing 12, "lights" an object using over 60 division of the angle of incidence, phi, and colors points that fall in different divisions different colors (modula 15). The results can tell you how the light is distributed over your object, and help you determine whether and where to move the light, and how many, and which, ranges of phi will be most effective. Illustration II.2 suggests how you can use testlighter to choose ranges of angle of incidence. Make a test marked rendering of the object, then select appropriate groups of divisions.

Testlighter requires a parameter file, 'tlparams' containing light position and object center.

Floor lighting code fragment, Listing 12-A , is lighting code for the patterned floor generating program given in Part I of this article. Add Listing 12-A and Listing 12-C to the 'zfloor' program (Listing 4.) of Part I. (Executable is included on disk.)

Cube lighting code fragment, Listing 12-B, lighting code for the cube generating program given in Part I of this article. Add Listing 12-B and Listing 12-C to the 'cuber' program (Listing 5.) of Part I. (Executable is included on disk.)

Lighting code fragment, Listing 12-C, is one possible division of phi's range, and treatment within each range, based on a complex, experimentally determined formula of remainders modula 27.

Lather, Listing 13, reads an outline image and generates an object of revolution. See Illustration II.5. The outline of the object should be upside down and in the upper left corner.

Justpal, Listing 14, reads and image and copies its palette to a small file called 'pal.' Several programs check to see if 'pal' is available in the directory. If it is, the program reads and loads the palette. Justpal provides an easy way to change the palette of an image.

Center.c, Listing 15, reads a file of object points and calculates min and max of x, y, and z values, and the center point. Use 'center.script' to run center on all the files of an object. Edit the resulting run_notes to produce a file with the centers of the object's sub-files. Run center again on this one file to get the center of the object. To find the center of a large group of objects make up a file with the center point of each object in the point file format of 'cc, xobj, yobj, zobj. You can use 'ed' to edit the ASCII point files. The center of an object is important for lighting in Part II.

Putting It Together

The programs give a choice of ways to create 3-D pictures.

1. You can create files of points for each object, then process these files thought coloring, placement, lighting, rendering to produce a 'rend' file (a rendered image) and 'zbuf' files.

2. You can string together code for all the processes and carry each point from generation through to rendering to produce a 'rend' file and 'zbuf' files all in one pass.

3. You can combine 'zbuf' files using compzbufs to get the final rendered image.

4. You can paste 'rend' files in front of and behind one another to produce the final rendered image.

5. All of the above.

Illustration II.4 shows the possibilities. The blossoms and leaves are four variations of one IFS vegetation object. Table II.3 gives the 'specs' file for the center plant. You will need the decoder program from my article "Make Your Own 3-D Vegetation Objects," AC's TECH Volume 3, Number 1, to decode the 'specs' file. I changed my decoder program to output a separate file for each blossom. This is easy since each blossom is a separate file and it comes with its own separate color. Sending points of different colors to different output files separates the flower's blossoms.

Each blossom point was then read, positioned, expanded to seven points to give it more body, colored, lighted, its shadow generated, and blossom and shadow rendered, all in one pass. The code that does this is not included but all of the necessary functions code is.

Using 'addpoints' I added a few points to the leaf files. The leaf points for a flower were lit as a single object. They were not expanded.

The flower pots were generated separately from formulas for points on a line and rotations, code that later evolved into the 'lather' program, Listing 13. The 'lather' program will produce pots from the object of revolution outline given in Illustration II.5.(d)

All the 'zbuf' files for the above were then submitted to 'compzbufs' which produced a rendered image of each potted flower and its shadow. I could make each flower and the empty pots separately since they do not intermingle.

The ground was produced using the 'zfloor' program from Parts I and II. The wall was produced using the lighting code of Listing 12-C in the 'lighter' program, Listing 10. I used the backer program, Listing 7, from Part I to put the wall behind the ground. Then, in *DPaint*, I picked up the flower elements and the empty pots as brushes and pasted them onto the ground and wall image. Positioning the brushes correctly was tricky. It could better have been done with the backer program. In DPaint I removed three or four stray plant points from the wall and filled in a blank line artifact that appeared midway up the flowers.

As you work on a 'paintings' you will think of effects you would like to implement. Customize one of the included programs to get the effect or use the two-point form to develop your own special algorithm. Your accumulated customized special tools will give your work personal style.

Biographical sketch

Laura M. Morrison has a master's degree in mathematics from New York University, NY. She has worked as an Operations Research Analyst designing applications software for Esso R&E, Union Carbide, and Eastern Airlines.

Ms. Morrison studied painting at the Art Student's League in New York and the Academy Julian in Paris.

✓

Listing 9

```
Morrison:PartII/PROGS/Listing_9
/* colorer.c Listing 9
 Assigns color id to an object. Assigns
 color register to each point accordingly.
 Copyright 1993 by Laura M. Morrison */
#include "stdio.h"
#include "libraries/dosextens.h"
#include "exec/exec.h"
#include "math.h"
/* map1[] assigns sixteen color registers to forty-six
color ids for points having xobj + yobj = odd. Colors 0
through 15 are solids. Colors 16 to 45 are checkered. */
 static int map1[] = { 0, 1, 2, 3, 4, 5, 6, 7,
                8, 9, 10, 11, 12, 13, 14, 15,
    1, 2, 3,   1, 2, 3,   1, 2, 3,   1, 2, 3,
               4, 5, 6,   4, 5, 6,   4, 5, 6,
              7, 8, 9,   7, 8, 9,   10, 11, 12  };
/* map2[] assigns color register to points
 having xobj + yobj = even and color id > 15 */
 static int map2[] = { 0, 1, 2, 3, 4, 5, 6, 7,
                8, 9, 10, 11, 12, 13, 14, 15,
  4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
            7, 8, 9, 10, 11, 12, 13, 14, 15,
        10, 11, 12, 13, 14, 15, 13, 14, 15 };
```

```
char outfile[120];
char infile[120];
static int cin, xin, yin, zin;
static int num, cout, xout, yout, zout;
main(argc, argv)
int argc;
char *argv[];
 {
   int i,id, mix, ptcount;
   int dm = 9999;
   FILE *fopen(), *dp, *sp, *lp, *pp;
    if ( argc < 2 )
    {
     printf("Need name of object.\n");
      exit(1);
    }
  if ((pp = fopen("c:params", "r"))==NULL)
   {
   printf(" Need a 'c:params' parameter file with \n");
   printf(" new color 'id' for object. \n");
   exit(1);
    }
   fscanf(pp," %d ",&id);
   fclose(pp);
  printf("\n\n\n\n\n\n\n\n");
   printf("              ");
  printf("ASSIGNS COLOR REGISTERS to OBJECT'S");
  printf("POINTS\n\n            ");
  printf("CORRESPONDING TO OBJECT'S COLOR ID\n");
   printf("              ");
  printf("Copyright 1991 by Laura M. Morrison\n\n\n");
   printf("              ");
  printf("Now processing file %s\n",argv[1]);
  printf("\n\n\n\n\n\n\n");
  sp = fopen(argv[1],"r");
   if( sp == NULL)
    {
  printf("Trouble opening input points file.\n");
   exit(2);
    }
  strcpy(outfile,"ram:");
  strcat(outfile,argv[1]);
  strcat(outfile,"c1");
  dp = fopen(outfile,"w");
   ptcount = 0;
readmore:
   fscanf(sp," %d %d %d %d ",&cin,&xin,&yin,&zin);
    if( cin = 997 )
    {
       goto finish;
    }
   mix = (xin + yin) % 2;
   if (mix==0)
  {
    cout = map1(id);
    }
   else
    {
    cout = map2(id);
    }
  fprintf(dp," %d %d %d %d \n",cout,xin,yin,zin);
   ptcount++;
   goto readmore;
finish:
    if(dp)
    {
     fprintf(dp," %d %d %d %d \n",dm,dm,dm,dm);
      fclose(dp);
    }
   if(sp) fclose(sp);
  lp = fopen("st_run_notes", "a");
   if (lp != NULL)
    {
     fprintf(lp," %s %s %s\n",
                 argv[0],argv[1],outfile);
     fprintf(lp," color id = %d \n",id);
      fclose(lp);
    }
}  /* end of main */
```

Listing 10

```
Morrison,part2,AVX070,Listing_10
/* Listing 10 lighter.c
 Copyright 1990 by Laura M. Morrison
 Reads object's points and colors them
 according to light. */
#include "stdio.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "exec/exec.h"
#include "math.h"
#define MAXPOINTS  4000
static float zone[] = {  0.0,
  0.02618, 0.05236, 0.07854, 0.10472,
 0.13090, 0.15708, 0.18326, 0.20944,
  0.23562, 0.26180, 0.28798, 0.31416,
  0.34034, 0.36652, 0.39270, 0.41888,
  0.44506, 0.47124, 0.49742, 0.52360,
  0.54978, 0.57596, 0.60214, 0.62832,
  0.65450, 0.68068, 0.70686, 0.73304,
  0.75922, 0.78540, 0.81158, 0.83776,
  0.86394, 0.89012, 0.91630, 0.94248,
  0.96866, 0.99484, 1.02102, 1.04720,
  1.07338, 1.09956, 1.12574, 1.15192,
  1.17810, 1.20428, 1.23046, 1.25664,
  1.28282, 1.30900, 1.33518, 1.36136,
  1.38754, 1.41372, 1.43990, 1.46608,
 1.49226, 1.51844, 1.54462, 1.57080  };
char outfile[120];
char infile[120];
char str[130];
static int xlit, ylit, zlit;
static int cobj, xobj, yobj, zobj;
static int xcen, ycen, zcen;
static int xoff, yoff, zoff;
static float cosalph1,cosalph2;
static float cosbeta1, cosbeta2, cosgam1, cosgam2;
static float costheta, phi, theta;
static float dist1, dist2;
static float sqrx, sqry, sqrz;
static float delx1,dely1,delz1;
static float delx2,dely2,delz2;
main(argc, argv)
int argc;
char *argv[];
 {
  int xoct, yoct, zoct,  row, col, i, j, k;
   int lbtl, mix, zen, occ, cc;
   int xoff, yoff, zoff, colorid;
   int f, count;
   int dm, dum;
   FILE *fopen(),  *dp, *sp, *lp, *pp, *bp;
   dum = 999;
   dm = 999;
    if ( argc < 2 )
     {
```

```
  printf("Need filename of object's points. \n");
   exit(2);
   }
  if ((pp = fopen("lparams","r"))==NULL)
   {
  printf(" Need a 'lparams' file with light");
  printf(" position,\n");
  printf(" and center of object: xcen, ycen, zcen\n");
  printf(" object color, and offsets \n");
   exit(3);
   }
  fscanf(pp," %d %d %d ",&xlit, &ylit, &zlit);
  fscanf(pp," %d %d %d %d",&xcen,&ycen,&zcen,&colorid);
  fscanf(pp," %d %d %d ",&xoff,&yoff,&zoff);
   fclose(pp);
  printf("\n\n\n\n");
   printf("              ");
  printf("LIGHT AN OBJECT \n\n");
   printf("        ");
  printf("Copyright 1993 by Laura M. Morrison\n\n\n");
   printf("    ... processing file %s \n\n",argv[1]);
   sp = fopen(argv[1],"r");
  strcpy(outfile,argv[1]);
 strcat(outfile,"text");
   op = fopen(outfile,"a");
    xcen = xcen + xoff;
    ycen = ycen + yoff;
    zcen = zcen + zoff;
readmore:;
    fscanf(sp," %d  %d  %d %d\n",
                  &cobj,&xobj,&yobj,&zobj);
    if(cobj > 997)
     {
       fclose(sp);
       goto finish;
     }
    count++;
    xobj = xobj + xoff;
    yobj = yobj + yoff;
    zobj = zobj + zoff;
   delx1 = (float)(xcen-xobj);
   dely1 = (float)(ycen-yobj);
   delz1 = (float)(zcen-zobj);
    sqrx = delx1*delx1;
    sqry = dely1*dely1;
    sqrz = delz1*delz1;
    dist1 = sqrt( sqrx + sqry + sqrz );
    if(dist1 != 0)
    {
     cosalph1 = delx1/dist1;
      cosbeta1 = dely1/dist1;
      cosgam1 = delz1/dist1;
     delx2 = (float)(xlit-xobj);
     dely2 = (float)(ylit-yobj);
     delz2 = (float)(zlit-zobj);
      sqrx = delx2*delx2;
      sqry = dely2*dely2;
      sqrz = delz2*delz2;
      dist2 = sqrt( sqrx + sqry + sqrz );
      if(dist2 != 0)
       {
        cosalph2 = delx2/dist2;
        cosbeta2 = dely2/dist2;
         cosgam2 =  delz2/dist2;
      costheta= cosalph1*cosalph2 + cosbeta1*cosbeta2
                                  + cosgam1*cosgam2;
       theta = acos(costheta);
       phi = 3.14159 - theta;
     } /* dist2 was zero */
   } /* dist1 was zero */
  mix = (xobj + yobj);
  rem = mix % 2;
 if((phi >= 0.0)&&(phi < zone[20]))
   {
    coc = cobj-7;
   }
 else if((phi >= zone[20])&&(phi < zone[30]))
   {
     if(rem == 1)
        coc = cobj-1;
     else
        coc = cobj-2;
   }
 else if((phi >= zone[30])&&(phi < zone[40]))
   {
       coc = cobj-1;
   }
 else if((phi >= zone[40])&&(phi < zone[50]))
   {
     if(rem == 1)
        coc = cobj-1;
     else
        coc = cobj;
   }
 else if((phi >= zone[50])&&(phi < zone[60]))
   {
      coc = cobj;
   }
   else
     coc = cobj;
 fprintf(op," %d %d %d %d \n",coc,xobj,yobj,zobj);
  goto readmore;
finish:
  fprintf(op,"%d %d %d %d \n",dum,dum,dum,dum);
  fclose(op);
  lp = fopen("df1:Run_Notes","a");
   if (lp != NULL)
   {
  fprintf(lp,"\n\n %s %s \n",argv[0],argv[1]);
  fprintf(lp," Light position (%d,%d,%d)\n",
                                  xlit,ylit,zlit);
  fprintf(lp," Object center (%d,%d,%d)\n",
                                  xcen,ycen,zcen);
  fprintf(lp," Object color %d \n",colorid);
  fclose(lp);
   }
}  /* end of main */
```

Listing 11

```
Morrison:Part2:MYCCGP:Listing_11
/* textlite.c listing11
 Copyright 1993 by Laura M. Morrison
 Colors each lighting angle sector a different
 color for sixty sectors. */
 #include "stdio.h"
 #include "libraries/dos.h"
 #include "libraries/dosextens.h"
 #include "exec/exec.h"
 #include "math.h"
 #define MAXPOINTS  1000
```

```
static float zone[] = {  0.0,
  0.02618,  0.05236,  0.07854,  0.10472,
  0.13090,  0.15708,  0.18326,  0.20944,
  0.23562,  0.26180,  0.28798,  0.31416,
  0.34034,  0.36652,  0.39270,  0.41888,
  0.44506,  0.47124,  0.49742,  0.52360,
  0.54978,  0.57596,  0.60214,  0.62832,
  0.65450,  0.68068,  0.70686,  0.73304,
  0.75922,  0.78540,  0.81158,  0.83776,
  0.86394,  0.89012,  0.91630,  0.94248,
  0.96866,  0.99484,  1.02102,  1.04720,
  1.07338,  1.09956,  1.12574,  1.15192,
  1.17810,  1.20428,  1.23046,  1.25664,
  1.28282,  1.30900,  1.33518,  1.36136,
  1.38754,  1.41372,  1.43990,  1.46608,
  1.49226,  1.51844,  1.54462,  1.57080  };
char outfile[120];
char infile[120];
char str[120];
static int xlit, ylit, zlit;
static int cobj, xobj, yobj, zobj;
static int xcen, ycen, zcen;
static int xoff, yoff, zoff;
static float cosalph1,cosalph2;
static float cosbeta1, cosbeta2, cosgam1, cosgam2;
static float costheta, phi, theta;
static float dist1, dist2;
static float sqrx, sqry, sqrz;
static float delx1,dely1,delz1;
static float delx2,dely2,delz2;
main(argc, argv)
int argc;
char *argv[];
 {
   int xscr, yscr, zscr, row, col;
   int iphi, mix, sem, ccc, cc;
   int xoff, yoff, zoff;
   int r, count;
   int dm, dum, i, j, k;
   FILE *fopen(),  *op, *ip, *pp, *wp;
   dum = 999;
  dm = 999;
   if ( argc < 2 )
    {
  printf("Need filename of object's points. \n");
   exit(2);
    }
  if ((pp = fopen("tlparams","r")) == NULL)
    {
  printf("Need a 'tlparams' file with light point\n");
  printf(" and center of object; xcen, ycen, zcen\n");
  printf(" and offsets.\n");
   exit(3);
    }
  fscanf(pp," %d %d %d ",&xlit, &ylit, &zlit);
  fscanf(pp," %d %d %d ",&xcen,&ycen,&zcen);
  fscanf(pp," %d %d %d ",&xoff,&yoff,&zoff);
   fclose(pp);
  printf("\n\n\n\n");
   printf("      TEST LIGHT AN OBJECT \n\n");
   printf("         ");
  printf("Copyright 1991 by Laura M. Morrison\n\n");
   printf("    --- processing file %s \n\n",argv[1]);
  ip = fopen(argv[1],"r");
  strcpy(outfile,argv[1]);
  strcat(outfile,".typ");
 op = fopen(outfile,"a");
```

```
    xcen = xcen + xoff;
    ycen = ycen + yoff;
    zcen = zcen + zoff;
 readmore:
    fscanf(ip," %d %d %d %d\n",
&cobj,&xobj,&yobj,&zobj);
     if(cobj == 999)
      {
        fclose(ip);
        goto finish;
      }
    count++;
    xobj = xobj + xoff;
    yobj = yobj + yoff;
    zobj = zobj + zoff;
   delx1 = (float)(xcen-xobj);
   dely1 = (float)(ycen-yobj);
   delz1 = (float)(zcen-zobj);
    sqrx = delx1*delx1;
    sqry = dely1*dely1;
    sqrz = delz1*delz1;
    dist1 = sqrt( sqrx + sqry + sqrz );
    if(dist1 != 0)
     {
       cosalph1 = delx1/dist1;
      cosbeta1 = dely1/dist1;
       cosgam1 = delz1/dist1;
      delx2 = (float)(xlit-xobj);
      dely2 = (float)(ylit-yobj);
      delz2 = (float)(zlit-zobj);
       sqrx = delx2*delx2;
       sqry = dely2*dely2;
       sqrz = delz2*delz2;
       dist2 = sqrt( sqrx + sqry + sqrz );
        if(dist2 != 0)
         {
          cosalph2 = delx2/dist2;
          cosbeta2 = dely2/dist2;
          cosgam2 =  delz2/dist2;
          costheta = cosalph1*cosalph2 +
                  cosbeta1*cosbeta2 + cosgam1*cosgam2;
          theta = acos(costheta);
          phi = 3.14159 - theta;
        } /* dist2 was zero */
    }  /* dist1 was zero */
   if((phi >= 0.0)&&(phi < zone[0]))
       ccc = 1;
  else if((phi >= zone[0])&&(phi < zone[1]))
       ccc = 2;
  else if((phi >= zone[1])&&(phi < zone[2]))
      ccc = 3;
  else if((phi >= zone[2])&&(phi < zone[3]))
       ccc = 4;
  else if((phi >= zone[3])&&(phi < zone[4]))
       ccc = 5;
  else if ((phi >=zone[4])&&(phi < zone[5]))
       ccc = 6;
  else if ((phi >=zone[5])&&(phi < zone[6]))
       ccc = 7;
  else if ((phi >=zone[6])&&(phi < zone[7]))
       ccc = 8;
  else if ((phi >= zone[7])&&(phi < zone[8]))
       ccc = 9;
  else if ((phi >= zone[8])&&(phi < zone[9]))
       ccc = 10;
  else if ((phi >= zone[9])&&(phi < zone[10]))
       ccc = 11;
```

```
else if ((phi >= zone[10])&&(phi < zone[11]))
     ccc = 12;
else if ((phi >= zone[11])&&(phi < zone[12]))
     ccc = 13;
else if ((phi >= zone[12])&&(phi < zone[13]))
     ccc = 14;
else if((phi >= zone[13])&&(phi < zone[14]))
     ccc = 15;
else if((phi >= zone[14])&&(phi < zone[15]))
    ccc = 1;
else if((phi >= zone[15])&&(phi < zone[16]))
     ccc = 2;
else if((phi >= zone[16])&&(phi < zone[17]))
     ccc = 3;
else if((phi >= zone[17])&&(phi < zone[18]))
     ccc = 4;
else if((phi >= zone[18])&&(phi < zone[19]))
     ccc = 5;
else if((phi >= zone[19])&&(phi < zone[20]))
     ccc = 6;
else if((phi >= zone[20])&&(phi < zone[21]))
     ccc = 7;
else if((phi >= zone[21])&&(phi < zone[22]))
     ccc = 8;
else if((phi >= zone[22])&&(phi < zone[23]))
     ccc = 9;
else if((phi >= zone[23])&&(phi < zone[24]))
     ccc = 10;
else if((phi >= zone[24])&&(phi < zone[25]))
     ccc = 11;
else if((phi >= zone[25])&&(phi < zone[26]))
     ccc = 12;
else if((phi >= zone[26])&&(phi < zone[27]))
     ccc = 13;
else if((phi >= zone[27])&&(phi < zone[28]))
     ccc = 14;
else if((phi >= zone[28])&&(phi < zone[29]))
     ccc = 15;
else if((phi >= zone[29])&&(phi < zone[30]))
     ccc = 1;
else if((phi >= zone[30])&&( phi < zone[31]))
     ccc = 2;
else if((phi >= zone[31])&&( phi < zone[32]))
     ccc = 3;
else if((phi >= zone[32])&&( phi < zone[33]))
     ccc = 4;
else if((phi >= zone[33])&&( phi < zone[34]))
     ccc = 5;
else if((phi >= zone[34])&&( phi < zone[35]))
     ccc = 6;
else if((phi >= zone[35])&&( phi < zone[36]))
     ccc = 7;
else if((phi >= zone[36])&&( phi < zone[37]))
     ccc = 8;
else if((phi >=zone[37])&&( phi < zone[38]))
     ccc = 9;
else if((phi >= zone[38])&&( phi < zone[39]))
     ccc = 10;
else if((phi >=zone[39])&&(phi < zone[40]))
     ccc = 11;
else if((phi >=zone[40])&&(phi < zone[41]))
     ccc = 12;
else if((phi >=zone[41])&&(phi < zone[42]))
     ccc = 13;
else if((phi >= zone[42])&&(phi < zone[43]))
     ccc = 14;
else if((phi >= zone[43])&&(phi < zone[44]))
     ccc = 15;
else if((phi >= zone[44])&&(phi < zone[45]))
     ccc = 1;
else if((phi >= zone[45])&&(phi < zone[46]))
     ccc = 2;
else if((phi >= zone[46])&&(phi < zone[47]))
     ccc = 3;
else if((phi >= zone[47])&&(phi < zone[48]))
     ccc = 4;
else if ((phi >= zone[48])&&(phi < zone[49]))
     ccc = 5;
else if ((phi >= zone[49])&&(phi < zone[50]))
     ccc = 6;
else if((phi >= zone[50])&&( phi < zone[51]))
     ccc = 7;
else if((phi >= zone[51])&&( phi < zone[52]))
     ccc = 8;
else if((phi >= zone[52])&&( phi < zone[53]))
     ccc = 9;
else if((phi >= zone[53])&&( phi < zone[54]))
     ccc = 10;
else if ((phi >= zone[54])&&(phi < zone[55]))
     ccc = 11;
else if((phi >= zone[55])&&(phi < zone[56]))
     ccc = 12;
else if ((phi >= zone[56])&&(phi < zone[57]))
     ccc = 13;
else if ((phi >= zone[57])&&(phi < zone[58]))
     ccc = 14;
else if ((phi >= zone[58])&&(phi < zone[59]))
     ccc = 15;
else if ((phi >= zone[59])&&(phi < zone[60]))
     ccc = 1;
 else
     ccc = 2;
 fprintf(op," %d %d %d %d \n",ccc,xobj,yobj,zobj);
 goto readmore;
finish:;
 fprintf(op,"%d %d %d %d \n",dum,dum,dum,dum);
 fclose(op);
 lp = fopen("df1:Run_Notes","a");
  if (lp != NULL)
   {
 fprintf(lp,"\n\n %s %s \n",argv[0],argv[1]);
fprintf(lp," Light position (%d,%d,%d)\n",
                             xlit,ylit,zlit);
 fprintf(lp," Object center (%d,%d,%d)\n",
                             xcen,ycen,zcen);
  fclose(lp);
   }
}  /* end of main */
```

Listing 12A

```
Morrison.Part2.MYALIB.Listing_12A
/* zloop fragment Listing12A
Copyright 1993 by Laura M. Morrison *
/* Insert this code fragment in zloop
program given in Part 1 of this article. */
/* The lighter program works with a 'center'
point which it uses to calculate the normal
line to the object.  To get a normal line
perpendicular to the floor at each point of
```

```
the floor we take a fake "center" at each
point of the floor. The "center" at floor
[illegible]
- [illegible] -400, z[ir]. */
   xcen = x[i];
   ycen = -400;
   zcen = z[ir];
  delx1 = (float)(xcen-x[i]);
  dely1 = (float)(ycen-y[i]);
  delz1 = (float)(zcen-z[ir]);
  sqrx = delx1*delx1;
  sqry = dely1*dely1;
  sqrz = delz1*delz1;
  dist1 = sqrt( sqrx + sqry + sqrz );

  if(dist1 != 0)
    {
     cosalph1 = delx1/dist1;
    cosbeta1 = dely1/dist1;
     cosgam1 = delz1/dist1;
    delx2 = (float)(x[i]-x[i]);
    dely2 = (float)(y[i]-y[i]);
    delz2 = (float)(z[ir]-z[ir]);
    sqrx = delx2*delx2;
    sqry = dely2*dely2;
    sqrz = delz2*delz2;
    dist2 = sqrt( sqrx + sqry + sqrz );
     if(dist2 != 0)
      {
       cosalph2 = delx2/dist2;
       cosbeta2 = dely2/dist2;
       cosgam2 = delz2/dist2;
costheta = cosalph1*cosalph2 + cosbeta1*cosbeta2 +
                                         cosgam1*cosgam2;
       theta = acos(costheta);
       phi = 3.14159 - theta;
     } /* dist2 was zero */
   } /* dist1 was zero */
   r = count % 5;
  mix0 = 100*(x[i] + z[ir]);
  mix1 = 100*(x[i] + z[ir]) + -1;
  mix2 = 100*(x[i] + z[ir]) + -2;
  mix3 = 100*(x[i] + z[ir]) + -3;
  mix4 = 100*(x[i] + z[ir]) + -4;
 mix5 = 100*(x[i] + z[ir]) + -5;

/* end of code fragment */
```

Listing 12B

```
[illegible] Listing_12B
/* shader code fragment Listing 12B
Copyright 1994 by Laura M. Morrison
Object in cube1.c given in
Part I of this article. */
      xobj = (int)xobjf;
      yobj = (int)yobjf;
      zobj = (int)zobjf;
/* Again requires a "center" point
to calculate the normal to the
surface at the point being [illegible]
we want a "center" that will
give a line perpendicular to the side
of the cube at the point being
calculated. */
      if((i==1)||(i==1))
        {
          ncenx = xobjf;
          nceny = yobjf;
          ncenz = zcenf;
        }
      else if((i==3)||(i==4))
        {
         ncenx = xcenf;
         nceny = yobjf;
         ncenz = zobjf;
        }
      else
        {
         ncenx = xobjf;
         nceny = ycenf;
         ncenz = zobjf;
        }
     delx1 = (float)(ncenx-xobj);
     dely1 = (float)(nceny-yobj);
     delz1 = (float)(ncenz-zobj);
         sqrx = delx1*delx1;
         sqry = dely1*dely1;
         sqrz = delz1*delz1;
         dist1 = sqrt( sqrx + sqry + sqrz );
          if(dist1 != 0)
           {
             cosalph1 = delx1/dist1;
             cosbeta1 = dely1/dist1;
             cosgam1 = delz1/dist1;
            delx2 = (float)(xlit-xobj);
            dely2 = (float)(ylit-yobj);
            delz2 = (float)(zlit-zobj);
             sqrx = delx2*delx2;
             sqry = dely2*dely2;
             sqrz = delz2*delz2;
            dist2 = sqrt( sqrx + sqry + sqrz );
            if(dist2 != 0)
              {
               cosalph2 = delx2/dist2;
               cosbeta2 = dely2/dist2;
               cosgam2 = delz2/dist2;
               costheta = cosalph1*cosalph2 +
cosbeta1*cosbeta2 + cosgam1*cosgam2;
               theta = acos(costheta);
               phi = 3.14159 - theta;
            } /* dist2 was zero */
        } /* dist1 was zero */
         r = count % 5;
        mix0 = 100*(yobj + xobj);
        mix1 = 100*(yobj + xobj) + -1;
        mix2 = 100*(yobj + xobj) + -2;
        mix3 = 100*(yobj + xobj) + -3;
        mix4 = 100*(yobj + xobj) + -4;
        mix5 = 100*(yobj + xobj) + -5;
/* end of code fragment */
```

Listing 12C

THIS LISTING IS NOT COMPLETE AS SHOWN

```
Morrison/Part2.SPYDCCP.Listing12C
/* lighting fragment: Listing_12C
 Insert in all ccp-and-cuber programs given in
 Part 1 of this article. */
        if(r==0)
            rem = mix0 % 27;
        else if(r==1)
            rem = mix1 % 27;
        else if(r==2)
            rem = mix2 % 27;
        else if(r==3)
            rem = mix3 % 27;
        else
            rem = mix4 % 27;

      if((phi >= 0.0)&&(phi < zone[0] ))
        {
          if(rem >= 0)
             ccc = 1;
           else
             ccc = cobj/3;
        }
      else if((phi >= zone[0] )&&(phi < zone[1]))
        {
          if(rem >= 1)
            ccc = 1;
          else
            ccc = cobj-2;
        }
      else if((phi >= zone[1] )&&(phi < zone[2]))
        {
             ccc = cobj-2;
        }
      else if((phi >= zone[2] )&&(phi < zone[3]))
        {
             if(rem < 1 )
               ccc = cobj-1;
             else
               ccc = cobj-2;
        }
      else if((phi >= zone[3] )&&(phi < zone[4]))
        {
          if(rem < 2 )
             ccc = cobj-1;
          else
             ccc = cobj-2;
        }
      else if ((phi >=zone[4])&&(phi < zone[5]))
        {
          if(rem < 3)
             ccc = cobj-1;
          else
             ccc = cobj-2;
        }
      else if ((phi >=zone[5])&&(phi < zone[6]))
        {
           if(rem < 4)
              ccc = cobj-1;
           else
              ccc = cobj-2;
        }

else if ((phi >=zone[6])&&(phi < zone[7] ))
   {
      if(rem < 5)
        ccc = cobj-1;
      else
        ccc = cobj-2;
   }
else if ((phi >= zone[7])&&(phi < zone[8]))
   {
     if( rem < 6)
       ccc = cobj-1;
     else
       ccc = cobj-2;
   }
else if ((phi >= zone[8])&&(phi < zone[9]))
   {
     if( rem < 7)
      ccc = cobj-1;
     else
       ccc = cobj-2;
   }
else if ((phi >= zone[9])&&(phi < zone[10]))
   {
     if( rem < 8)
       ccc = cobj-1;
     else
       ccc = cobj-2;
   }
else if ((phi >= zone[10])&&(phi < zone[11]))
   {
     if( rem < 9)
       ccc = cobj-1;
     else
       ccc = cobj-2;
   }
else if ((phi >= zone[11])&&(phi < zone[12]))
   {
     if( rem < 10)
       ccc = cobj-1;
     else
       ccc = cobj-2;
   }
else if ((phi >= zone[12])&&(phi < zone[13]))
   {
     if( rem < 11)
       ccc = cobj-1;
     else
       ccc = cobj-2;
   }
else if((phi >=zone[13])&&(phi < zone[14]))
   {
      if(rem < 12)
        ccc = cobj-1;
      else
        ccc = cobj-2;
   }
else if((phi >= zone[14])&&(phi < zone[15]))
   {
      if(rem < 13)
        ccc = cobj-1;
      else
        ccc = cobj-2;
   }
else if((phi >= zone[15])&&(phi < zone[16]))
   {
      if(rem < 14)
        ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[16])&&(phi < zone[17]))
      {
        if(rem < 16)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[17])&&(phi < zone[18]))
      {
        if(rem < 16)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[18])&&(phi < zone[19]))
      {
        if(rem < 17)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[19])&&(phi < zone[20]))
      {
        if(rem < 18)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[20])&&(phi < zone[21]))
      {
        if(rem < 19)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[21])&&(phi < zone[22]))
      {
        if(rem < 20)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[22])&&(phi < zone[23]))
      {
        if(rem < 21)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }

    else if((phi >= zone[23])&&(phi < zone[24]))
      {
        if(rem < 22)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[24])&&(phi < zone[25]))
      {
        if(rem < 23)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
```

TABLE_2.1 Parameter files

Parameter file for colorer: 'clparams'

    6
    FORMAT:
            color id (%d) for object

Parameter file for lighter: 'lparams'

    100 800 900
    0 50 0 6
    320 250 1000
    FORMAT:
            Light positon: (xlit,ylit,zlit)
            Center of object: (xcen,ycen,zcen)
            Color id
            Offsets: xoff, yoff, zoff

Parameter file for testlighter: 'tlparams'

    100 800 900
    0 50 0
    320 250 1000
    FORMAT:
            Light position: (xlit,ylit,zlit)
            Center of object: (xcen,ycen,zcen)
            Offsets: xoff, yoff, zoff

Parameter file for lather: 'laparams'

    72 104
    1 3
    FORMAT:
            Width and height of input outline.
            Increment amount.
            Color id

```
    else if((phi >= zone[25])&&(phi < zone[26]))
      {
        if(rem < 24)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[26])&&(phi < zone[27]))
      {
        if(rem < 25)
          ccc = cobj-1;
        else
          ccc = cobj-2;
      }
    else if((phi >= zone[27])&&(phi < zone[28]))
      {
        if(rem < 26)
          ccc = cobj-1;
        else
          ccc = cobj-2;
```

Table 2.2 Script Files

```
Script for colorer:
  .key filename/a
  copy colorer to ram:
  copy uiparams to ram:
  cd ram:
  IF EXISTS <filename>0
  colorer <filename>0
   ENDIF
  IF EXISTS <filename>1
  colorer <filename>1
   ENDIF
  IF EXISTS <filename>2
  colorer <filename>2
   ENDIF
   ...
  IF EXISTS <filename>9
  colorer <filename>9
   ENDIF

Script for center:

  IF EXISTS <filename>0
  center <filename>0
   ENDIF
  IF EXISTS <filename>1
  center <filename>
   ENDIF
  IF EXISTS <filename>2
  center <filename>2
   ENDIF
      ...

  IF EXISTS <filename>9
  center <filename>9
   ENDIF

Script file for lighter:
  .key <filename>/a
  copy lighter to ram:
  copy uiparams to ram:
   cd ram:
  IF EXISTS <filename>0
  lighter <filename>0
   ENDIF
  IF EXISTS <filename>1
  lighter <filename>1
   ENDIF
  IF EXISTS <filename>2
  lighter <filename>2
   ENDIF
       ...

  IF EXISTS <filename>9
  lighter <filename>9
   ENDIF
   cd df1:

Script file for
testlighter:
  .key <filename>/a
  copy testlighter to ram:
  copy uiparams to ram:
   cd ram:
  IF EXISTS <filename>0
  testlighter <filename>0
   ENDIF
  IF EXISTS <filename>1
  testlighter <filename>1
   ENDIF
  IF EXISTS <filename>2
  testlighter <filename>2
   ENDIF
       ...

  IF EXISTS <filename>9
  testlighter <filename>9
   ENDIF
   cd df1:
```

```
   }
   else if((pb1 >= zone[26])&&(pb1 < zone[27]))
   {
      if(rem < 27)
         cbc = cbb+1;
      else
         cbc = cbb-1;
   }
   else if ((pb1 >= zone[27])&&(pb1 < zone[28]))
   {
      cbc = cbb+1;
   }
   else if((pb1 >= zone[30])&&(pb1 < zone[31]))
   {
      cbc = cbb-1;
   }
 else if((pb1 >= zone[1])&&(pb1 < zone[2]))
   {
      if(rem < 1 )
         cbc = cbb;
      else
```

```
         cbc = cbb-1;
   }
else if((pb1 >= zone[2])&&(pb1 < zone[3]))
   {
      if(rem < 2)
         cbc = cbb;
      else
         cbc = cbb-1;
   }
else if((pb1 >= zone[3])&&(pb1 < zone[4]))
   {
      if(rem < 3)
         cbc = cbb;
      else
         cbc = cbb-1;
   }
else if((pb1 >= zone[4])&&(pb1 < zone[5]))
   {
      if(rem < 4 )
         cbc = cbb;
      else
      cbc = cbb-1;
   }
else if((pb1 >= zone[25])&&(pb1 < zone[26]))
   {
      if(rem < 5 )
         cbc = cbb;
      else
         cbc = cbb-1;
   }
else if((pb1 >= zone[36])&&(pb1 < zone[37]))
   {
      if(rem < 6)
         cbc = cbb;
      else
         cbc = cbb-1;
   }
else if((pb1 >= zone[37])&&(pb1 < zone[38]))
   {
      if(rem < 7)
         cbc = cbb;
      else
         cbc = cbb-1;
   }
else if((pb1 >= zone[38])&&(pb1 < zone[39]))
   {
     if(rem < 8)
         cbc = cbb;
      else
         cbc = cbb-1;
   }
else if((pb1 >= zone[39])&&(pb1 < zone[40]))
   {
     if(rem < 39)
         cbc = cbb;
      else
         cbc = cbb-1;
   }
else if((pb1 >= zone[40])&&(pb1 < zone[41]))
   {
     if(rem < 10)
         cbc = cbb;
      else
         cbc = cbb-1;
   }
else if((pb1 >= zone[41])&&(pb1 < zone[42]))
   {
      if ( rem < 11)
         occ = cobj;
      else
         occ = cobj-1;
   }

else if((phi >= zone[42])&&(phi < zone[43]))
   {
      if(rem < 12)
         occ =  cobj;
      else
         occ = cobj-1;
   }
else if((phi >= zone[43])&&(phi < zone[44]))
   {
      if(rem < 13)
         occ =  cobj;
      else
         occ = cobj-1;
   }
else if((phi >= zone[44])&&(phi < zone[45]))
   {
      if(rem < 14)
         occ =  cobj;
      else
         occ = cobj-1;
   }
else if((phi >= zone[45])&&(phi < zone[46]))
   {
      if(rem < 15)
         occ =  cobj;
     else
         occ = cobj-1;
   }
else if((phi >= zone[46])&&(phi < zone[47]))
   {
      if(rem < 16)
         occ =  cobj;
      else
         occ = cobj-1;
   }
else if((phi >= zone[47])&&(phi < zone[48]))
   {
      if(rem < 17)
         occ =  cobj;
      else
         occ = cobj-1;
   }
else if ((phi >= zone[48])&&(phi < zone[49]))
   {
      if(rem < 18)
         occ = cobj;
      else
         occ = cobj-1;
   }
else if ((phi >= zone[49])&&(phi < zone[50]))
  {
      if(rem < 19)
         occ = cobj;
      else
         occ = cobj-1;
   }
else if((phi >= zone[50])&&( phi < zone[51]))
   {
```

THIS LISTING IS NOT COMPLETE AS SHOWN

TABLE 2.3 SPECIFICATIONS FOR FLOWER

(See Volume 3 Issue 1 for 3-D_IFS_decoder Program.)

```
1.2 100  1.2 0  1.2 100
0.0 -160.0 0.0   0.4 0.4 0.4  0.0 0.0 0.0
160.0  350.0 160.0
0.0 -100.0 0.0   0.4 0.4 0.4  0.0 0.0 0.9
160.0  350.0 160.0
0.0 -100.0 0.0   0.4 0.4 0.4  0.0 0.0 -0.9
160.0  350.0 160.0
0.0 -100.0 0.0   0.4 0.4 0.4  0.9 0.0 0.0
160.0  350.0 160.0
0.0 -100.0 0.0   0.4 0.4 0.4  -0.9 0.0 0.0
160.0  350.0 160.0
0.0 -130.0 0.0   0.4 0.4 0.4  0.7 0.0 0.7
160.0  350.0 160.0
0.0  -130.0 0.0  0.4 0.4 0.4  -0.7 0.0 0.7
160.0  350.0  160.0
0.0  -130.0 0.0  0.4 0.4 0.4  0.7 0.0 -0.7
160.0  350.0  160.0
0.0  -130.0 0.0  0.4 0.4 0.4  -0.7 0.0 -0.7
160.0  350.0  160.0
0.0   0.0 0.0   0.0 0.4 0.0  0.0 0.0 0.0
160.0  350.0  160.0
999.9 999.9 999.9 999.9 999.9 999.9 999.9 999.9 999.9
999.9 999.9 999.9 999.9 999.9 999.9 999.9 999.9
```

The remainder of the source code and listings for this article can be found on the AC's TECH disk.

Please write to:
Laura Morrison
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722

Coming Up:

AC's TECH 4.1

GaugeClass—A class of gauge images for intuition. Article is a tutorial on the Amiga's object oriented features contained in BOOPSI.

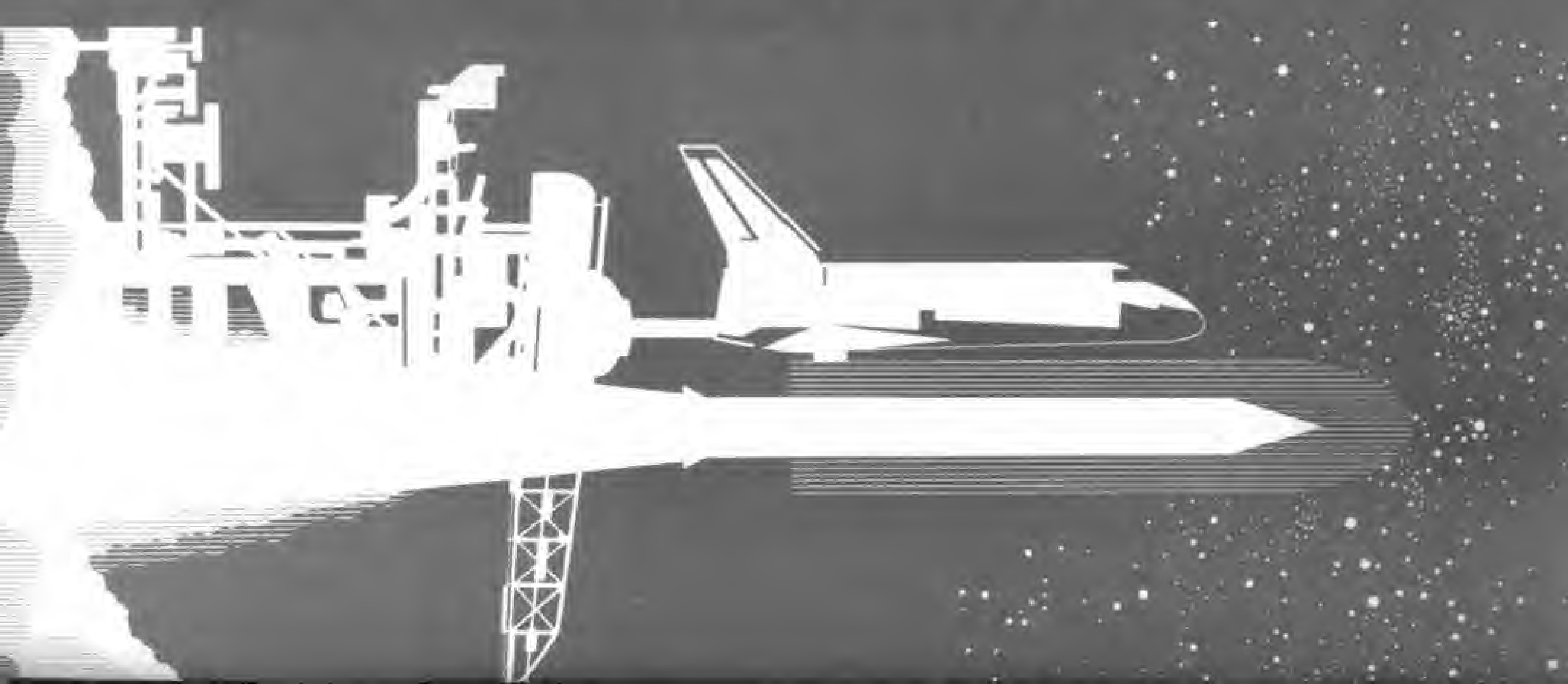
Huge Numbers—Part I of a series exploring an alternate floating point number system.

Complex Functions in Assembly—A follow-up to Bill Nee's series, Programming the Amiga in Assembly Language.

A Better Way to C—Exploring the use of the C++ programming language as a better version of C.

Also: Results of the AC's TECH Programming Contest!

DON'T MISS AN ISSUE!



TWO NEW RELEASES

FAST AND POWERFUL PRODUCTS FOR AREXX

Compile your ARexx programs with the REXX PLUS COMPILER and they will execute up to 18 times faster. The Intuition Interface allows even the most novice user to execute their programs at warp speed. Explicit error messages make debugging a breeze. The REXX PLUS COMPILER generates a listing that is easier to read than the original source. The listing contains nesting levels, flagged comments, a symbol table and a complete cross reference. Version 1.3 is a major upgrade that generates 40 to 60% smaller programs. All REXX RAINBOW LIBRARY SERIES functions can be included as part of the language.

Don't just take our word for it, here is what some of the experts have to say about the REXX PLUS COMPILER...

"...A SIGNIFICANT NEW PRODUCT WHICH ALL AREXX PROGRAMMERS SHOULD HAVE."
Amazing Computing, June 1992

"...THE AUTHORS HAVE IT RIGHT... IT COULD WELL BE A FUTURE AMIGA CLASSIC."
Amiga Computing UK, November 1992

"...IS A WELL-DESIGNED UTILITY THAT DOES ITS UTMOST TO SUPPORT THE COMPLETE AREXX ENVIRONMENT IN A TRANSPARENT FASHION."
Amiga World, September 1992

"...DOES THE JOB AND DOES IT WELL, EVEN ELEGANTLY."
Jump Disk, June 1992

NEW

REXX *Rainbow Library* SERIES

The REXX RAINBOW LIBRARY SERIES is a complete product line of support libraries designed specifically for use with ARexx. Each volume in the Series contains functions dedicated to a specific subject. The first volume in the series is the Stem/Array functions. It provides over 100 functions to manipulate single dimension arrays, which simplify ARexx arrays, Compound Symbols, Pointers and Subscripts. The functions include string manipulation, mathematical and scientific calculations and file access. Also included is the AssgnArray() function which assigns/retrieves arrays from/to other ARexx programs. With this function you can build your own single or multiple dimension array functions. Tutorials and examples are used throughout the manual. The REXX RAINBOW LIBRARY SERIES requires ARexx and works with or without REXX PLUS.

DEMO DISK AVAILABLE

Dineen Edwards Group
19785 W. 12 Mile Rd., Suite 305
Southfield, MI 48076-2553
313-352-4288

Amiga Dos is a registered trademark of Commodore Business Machine. ARexx is a registered trademark of Wishful Thinking.

IT'S SHIPPING!

BRILLIANCE

Professional Paint & Animation

DIGITAL CREATIONS